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 falling 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**

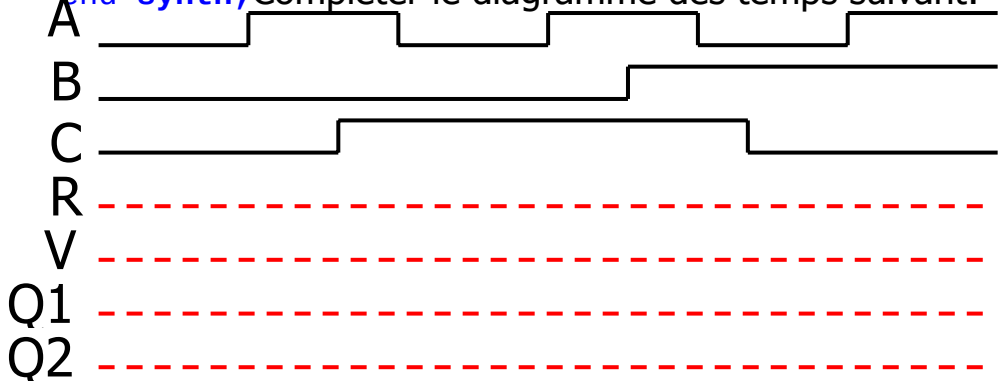
```
library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

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;
```

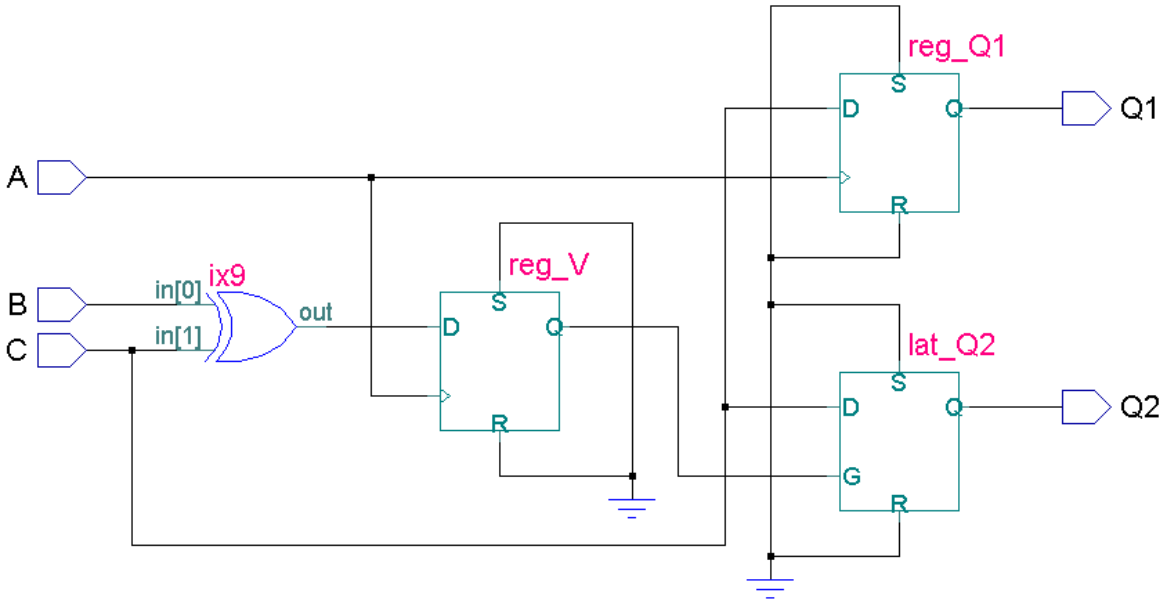
```
  CompZero: process (CountContenu)
  begin
    Zero <= '0';
    if (CountContenu = 0) then
      Zero <= '1';
    end if;
  end process;

end synth;
```

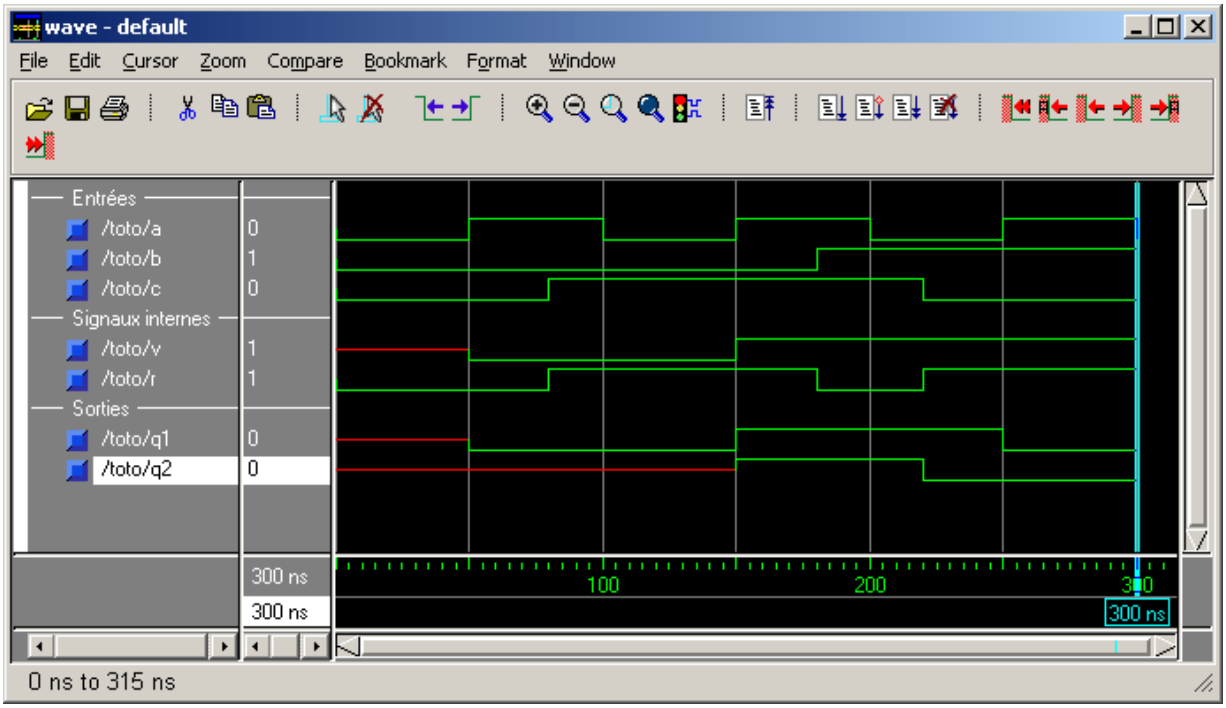
Dessiner le système représenté par le code VHDL suivant :

```
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:

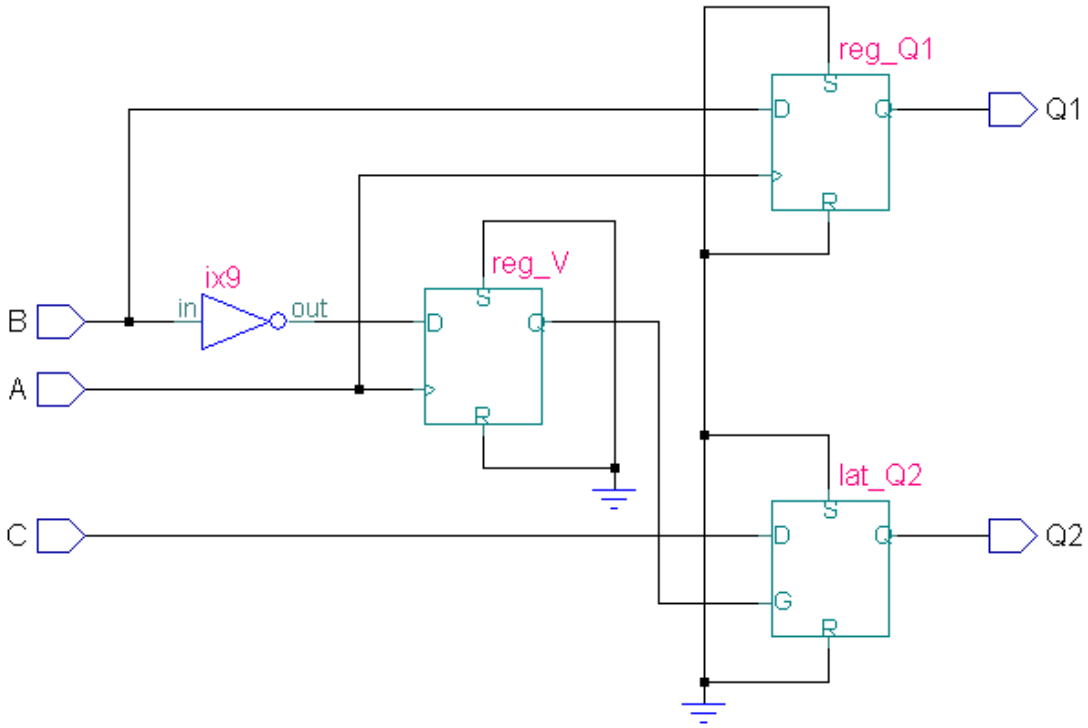



reg_Q1
S
D
Q
Q1
R
A
B
C
in[0]
in[1]
ix9
out
reg_V
S
D
Q
R
lat_Q2
S
D
Q
G
R
Q2


wave - default
File Edit Cursor Zoom Compare Bookmark Format Window
Entrées
/toto/a
/toto/b
/toto/c
Signaux internes
/toto/v
/toto/r
Sorties
/toto/q1
/toto/q2
0
1
0
1
1
0
0
300 ns
300 ns
100
200
300 ns
0 ns to 315 ns

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

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,…,X_0, et huit bits de sortie, Q_7,…,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:

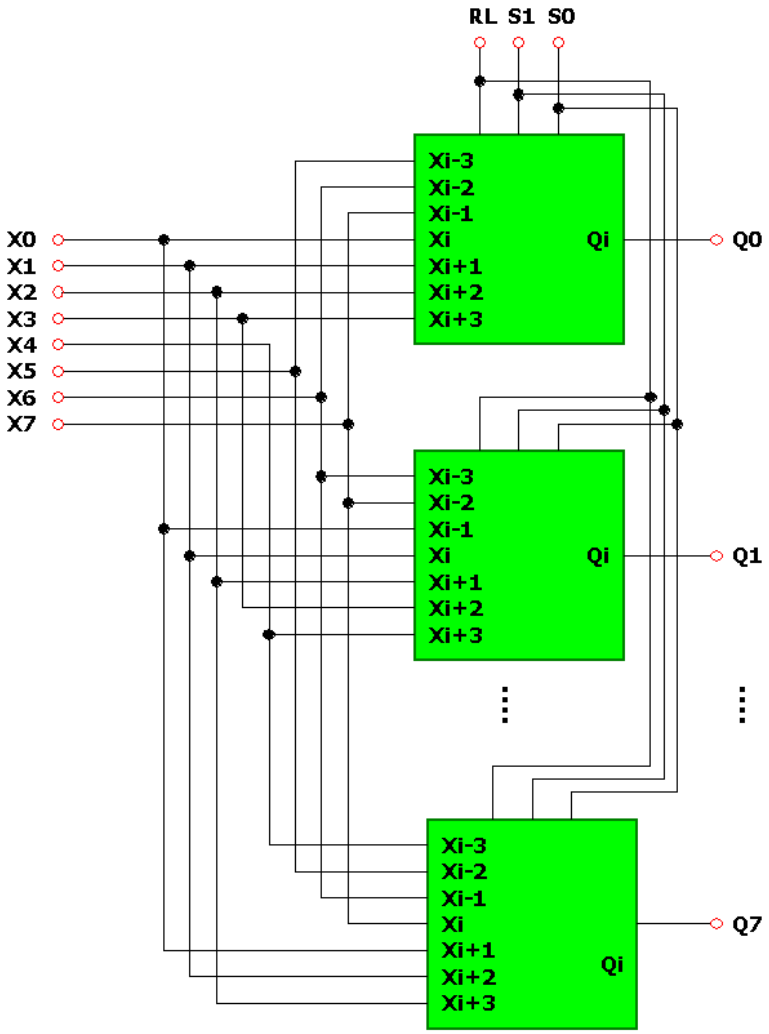
```
component mux_2x1
  port (
    D0 : in std_logic;
    D1 : in std_logic;
    S  : in std_logic;
    Q  : out std_logic); -- D0 si S=0; D1 sinon
end component;
```

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

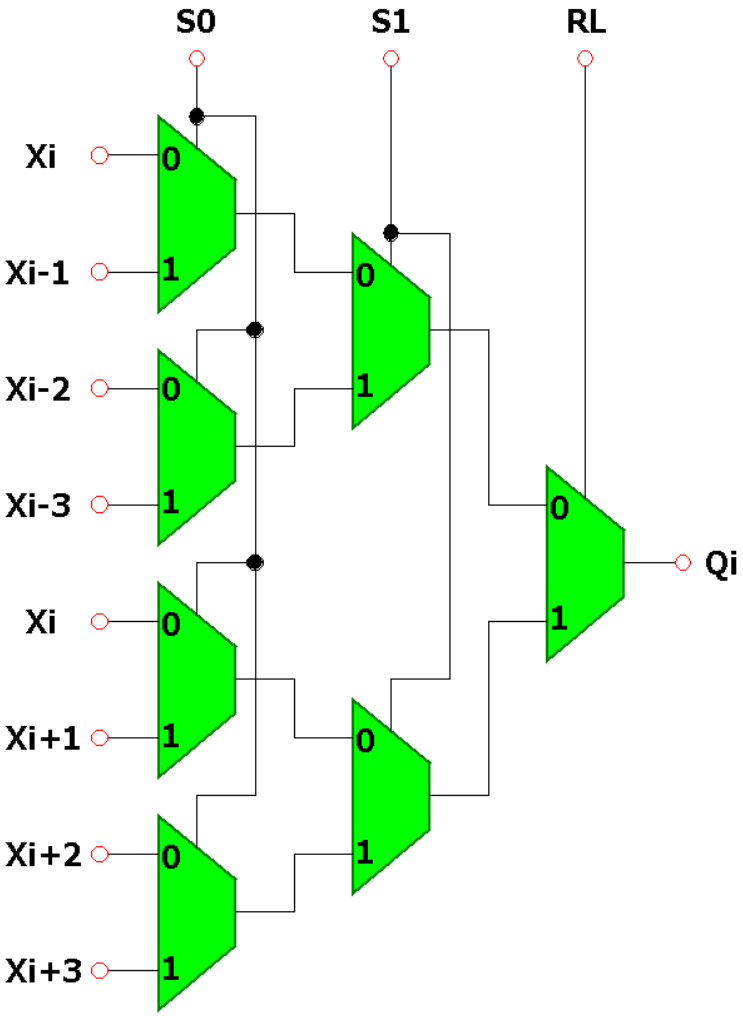
Tableau 1

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:*

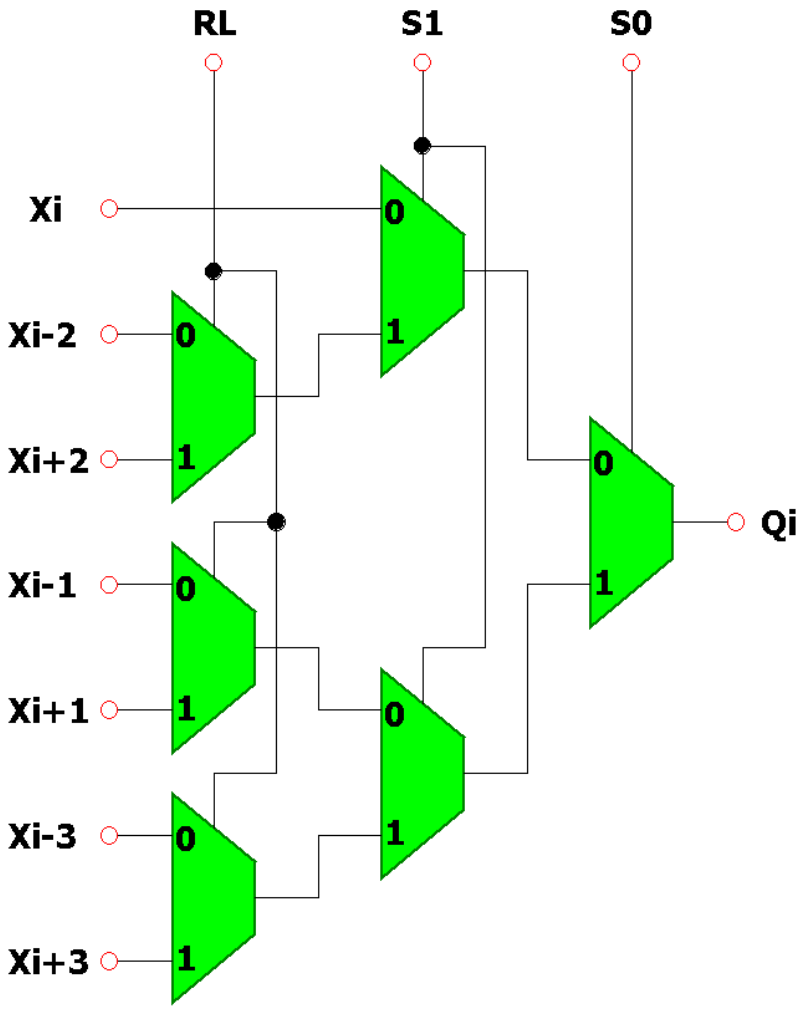


❖ *Schéma logique d'un composant:*



```
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;
```

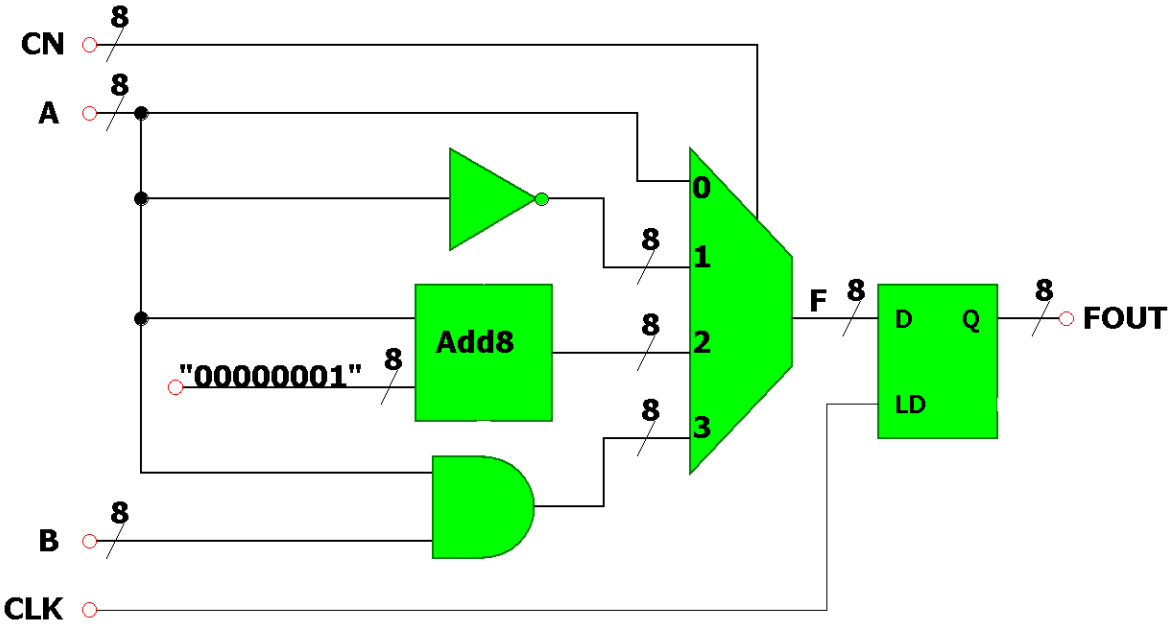
❖ *Schéma logique optimisé d'un composant:*



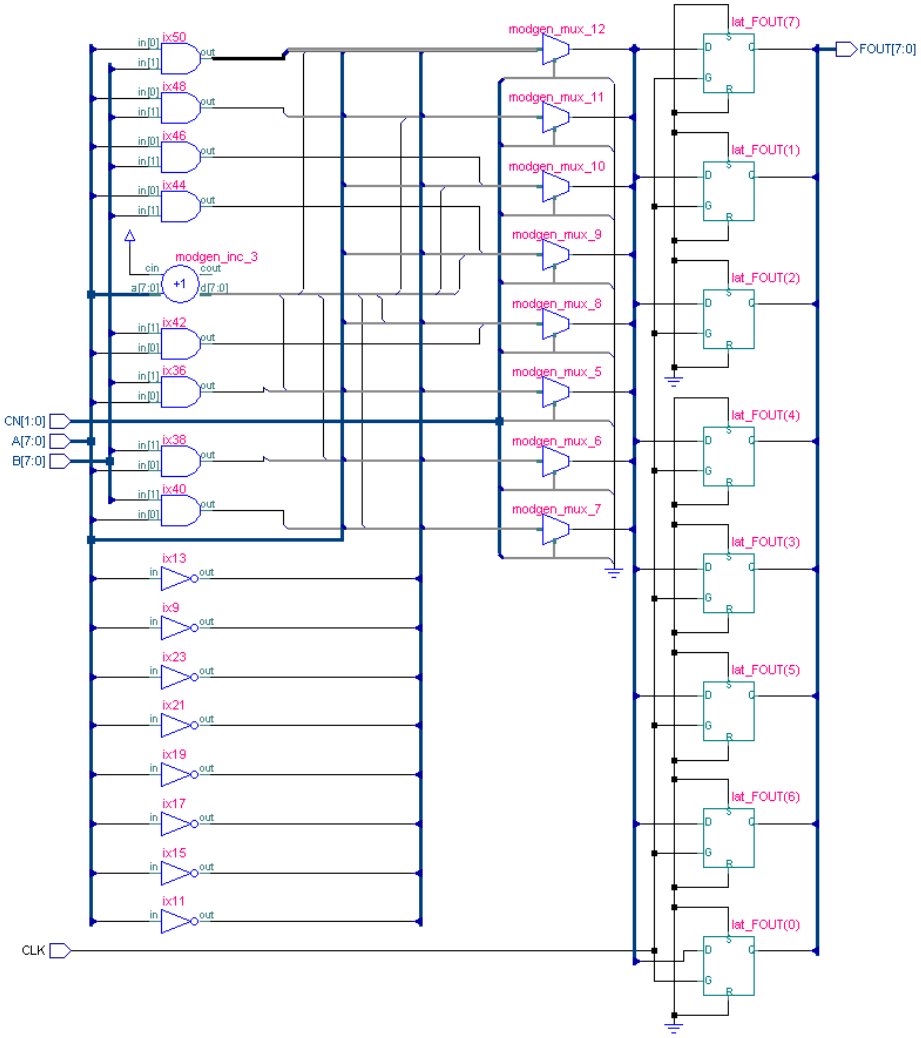
Dessiner le système représenté par le code VHDL suivant:

```
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 :

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

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.

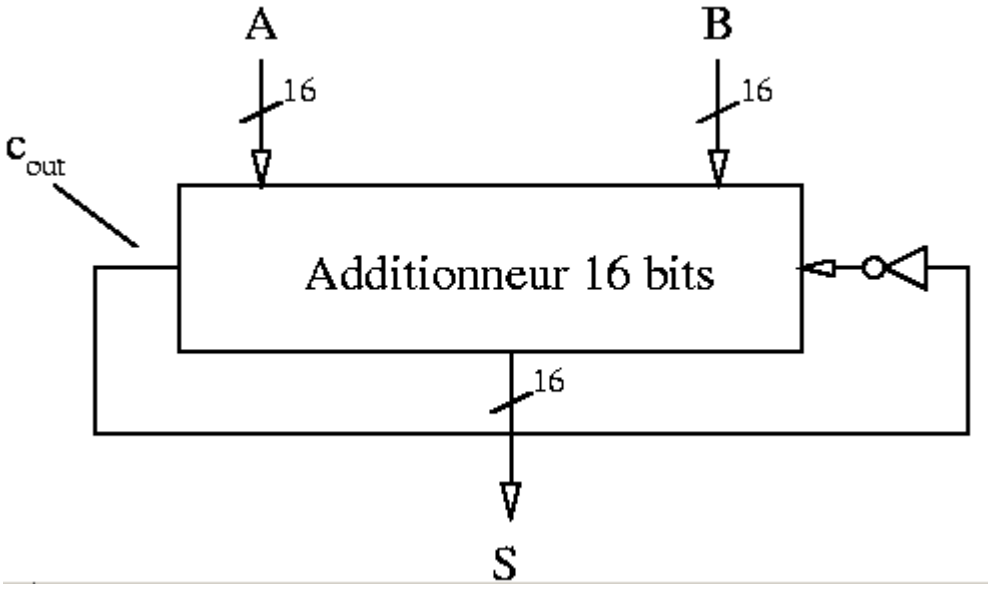


Figure 1

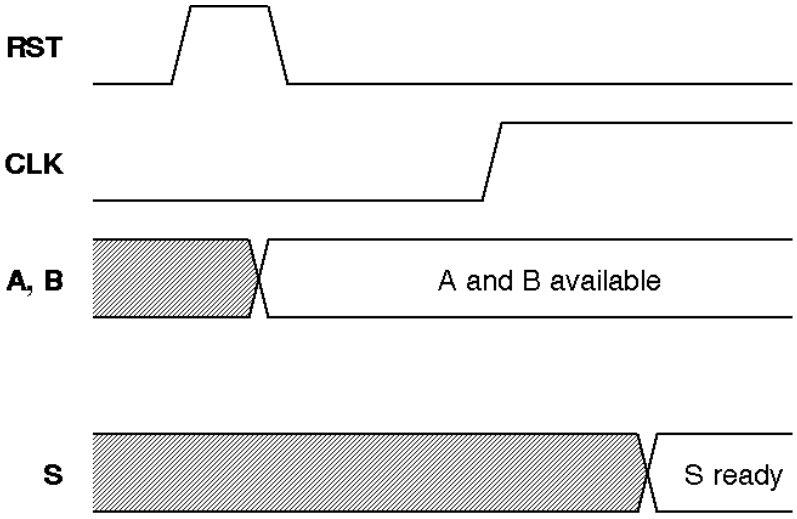


Figure 2

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

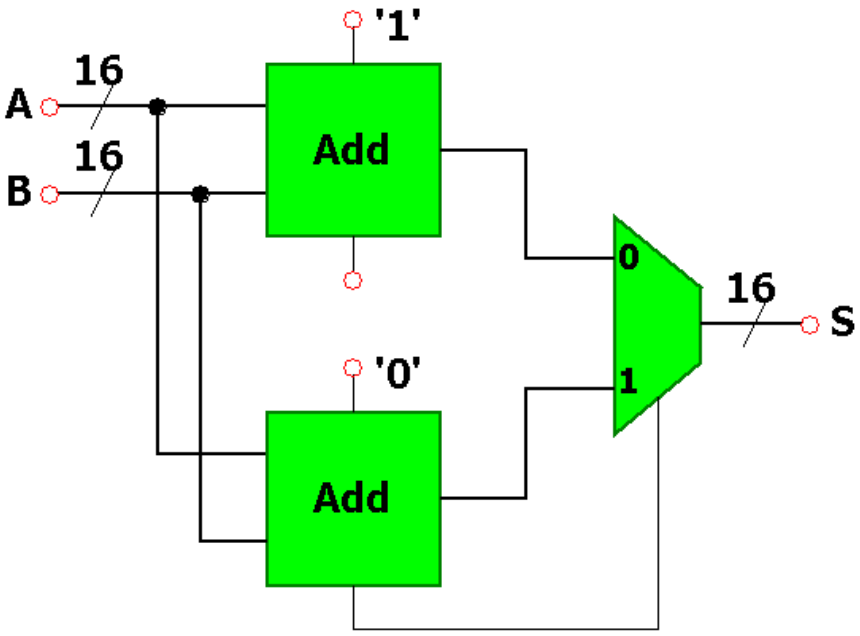
- *Pour Cin=0, Cout=0 (ce qui implique que Cin va passer à 1)*
- *Pour Cin=1, Cout=1 (ce qui implique que Cin va passer à 0)*

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

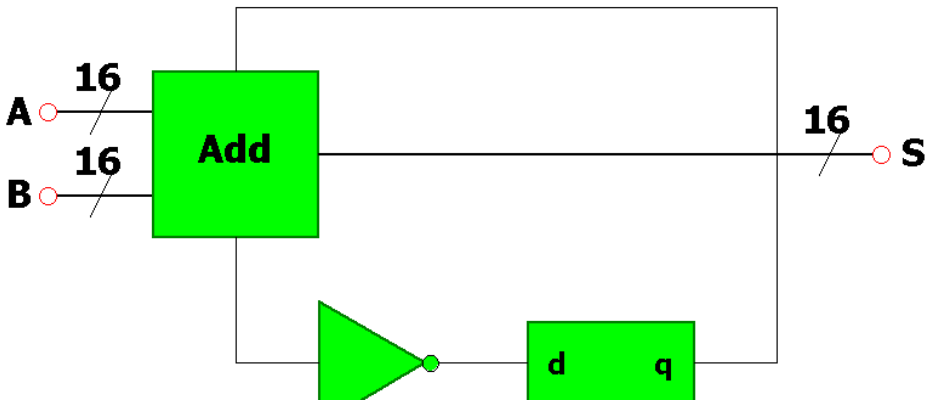
Exemple: A = 1111'1111h B = 0000'0000h

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;
```

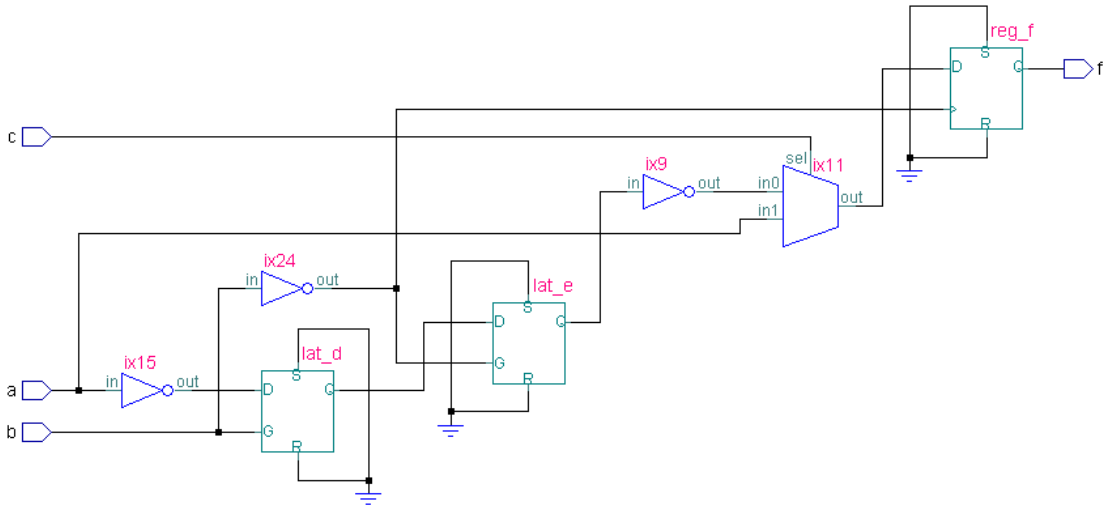
Considérer le programme VHDL de la figure 1.

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).

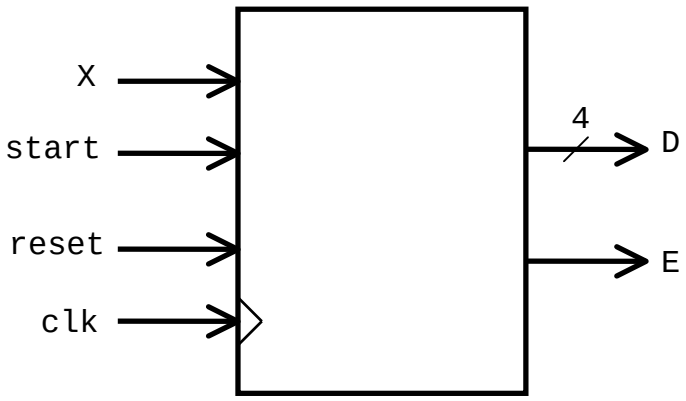


Figure 2

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).

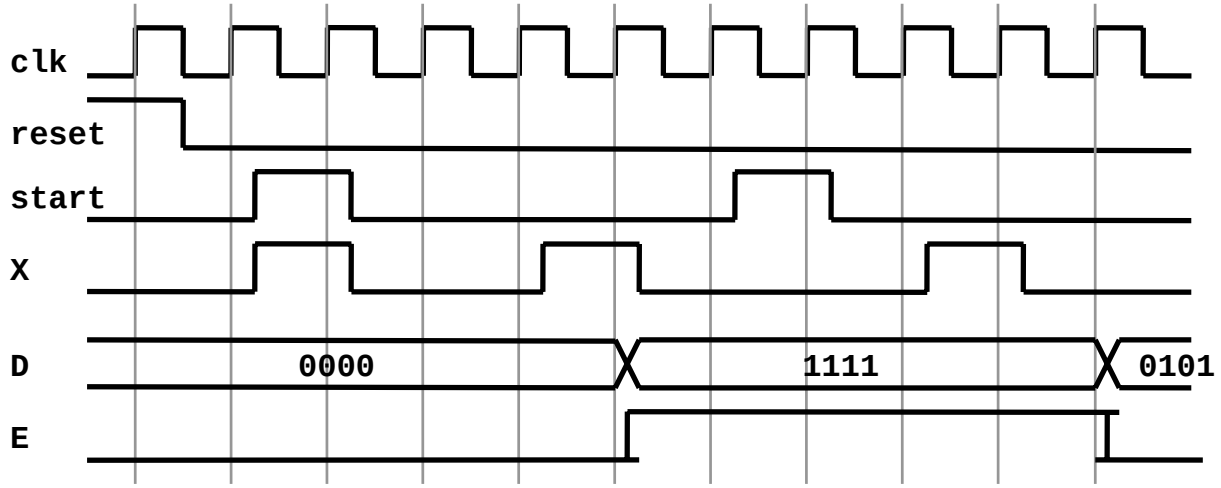


Figure 3

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

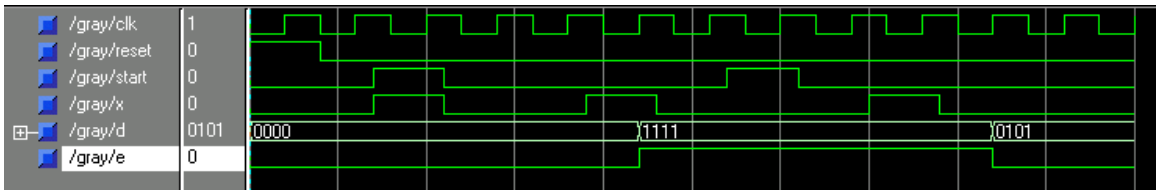
```
library ieee;
use ieee.std_logic_1164.all;

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:

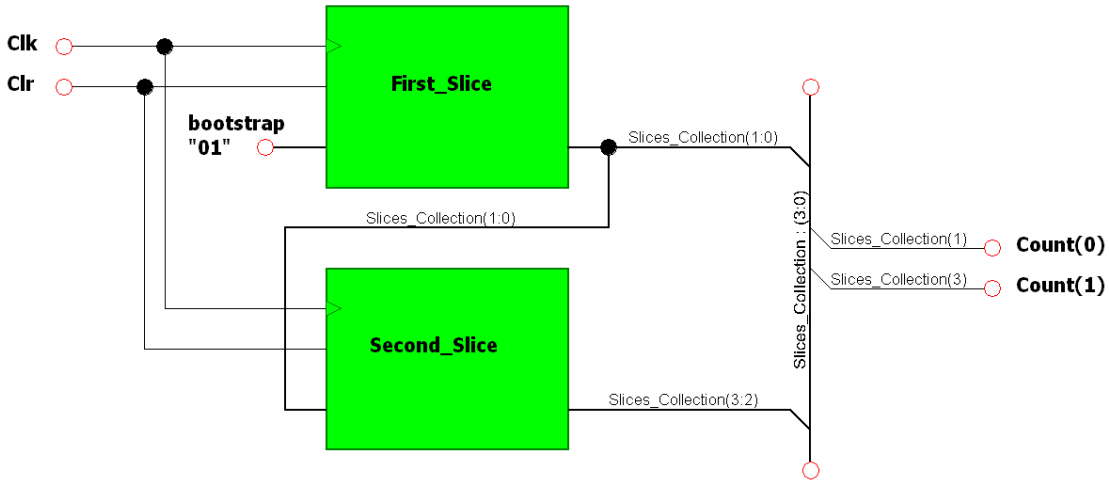
```
library ieee;
use ieee.std_logic_1164.all;

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
```

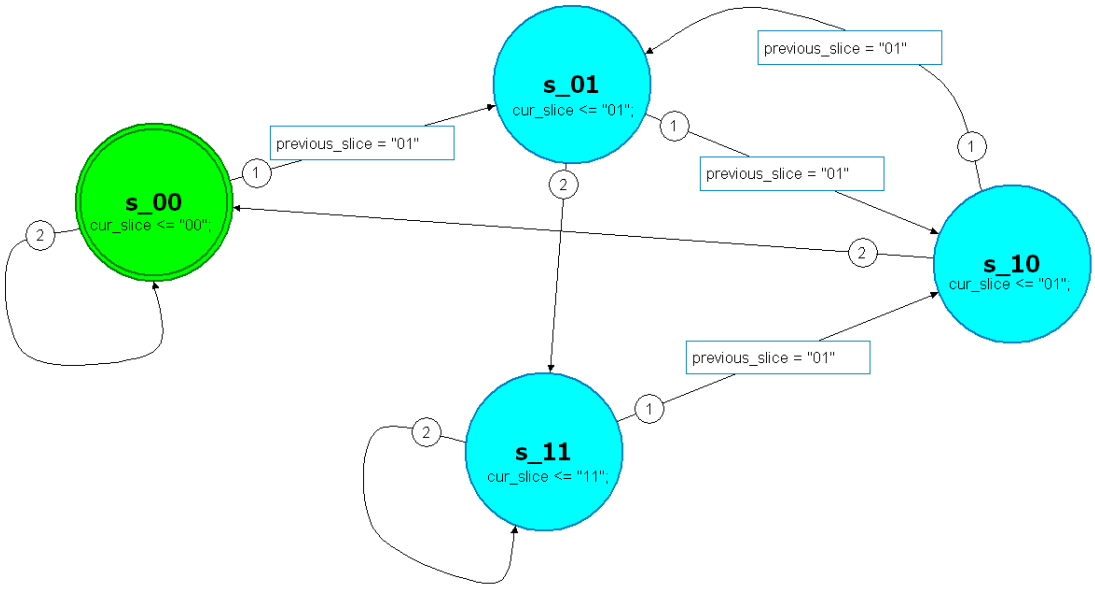
```
    port (clk            : in std_logic;
          reset          : in std_logic;
          previous_slice : in std_logic_vector(1 downto 0);
          slice          : out std_logic_vector(1 downto 0));
  end component;

  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.

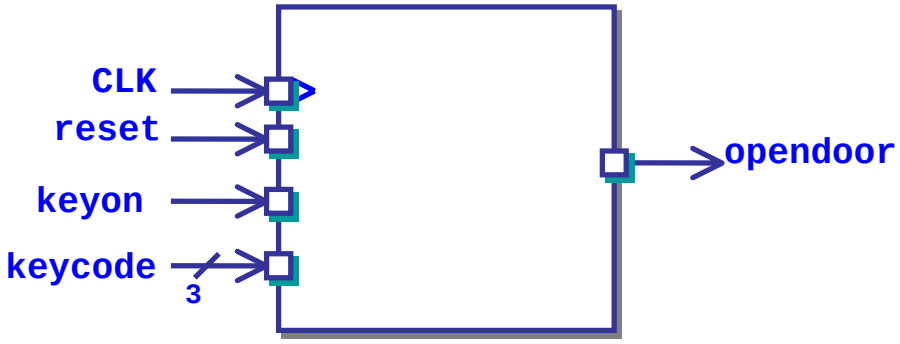


c. First_Slice: 00 -> 01 -> 10 -> 01 -> 10 ->

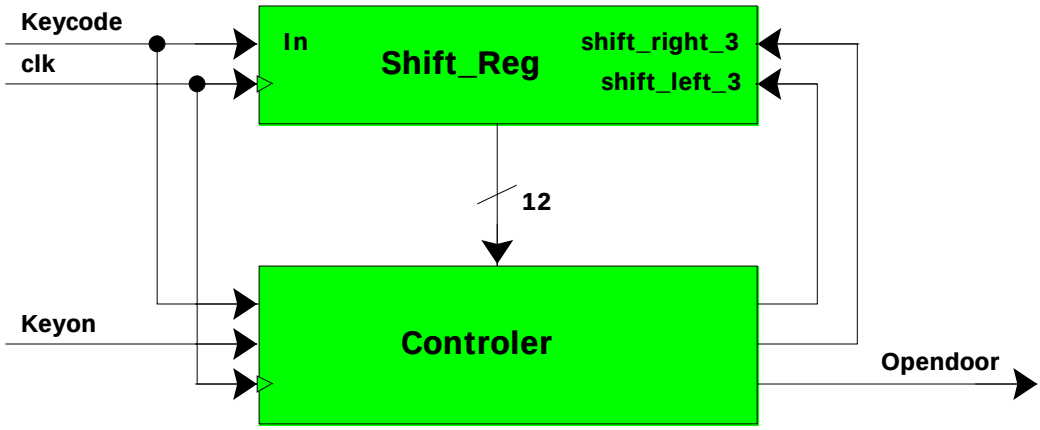
Second_Slice: 00 -> 00 -> 01 -> 11 -> 10 ->

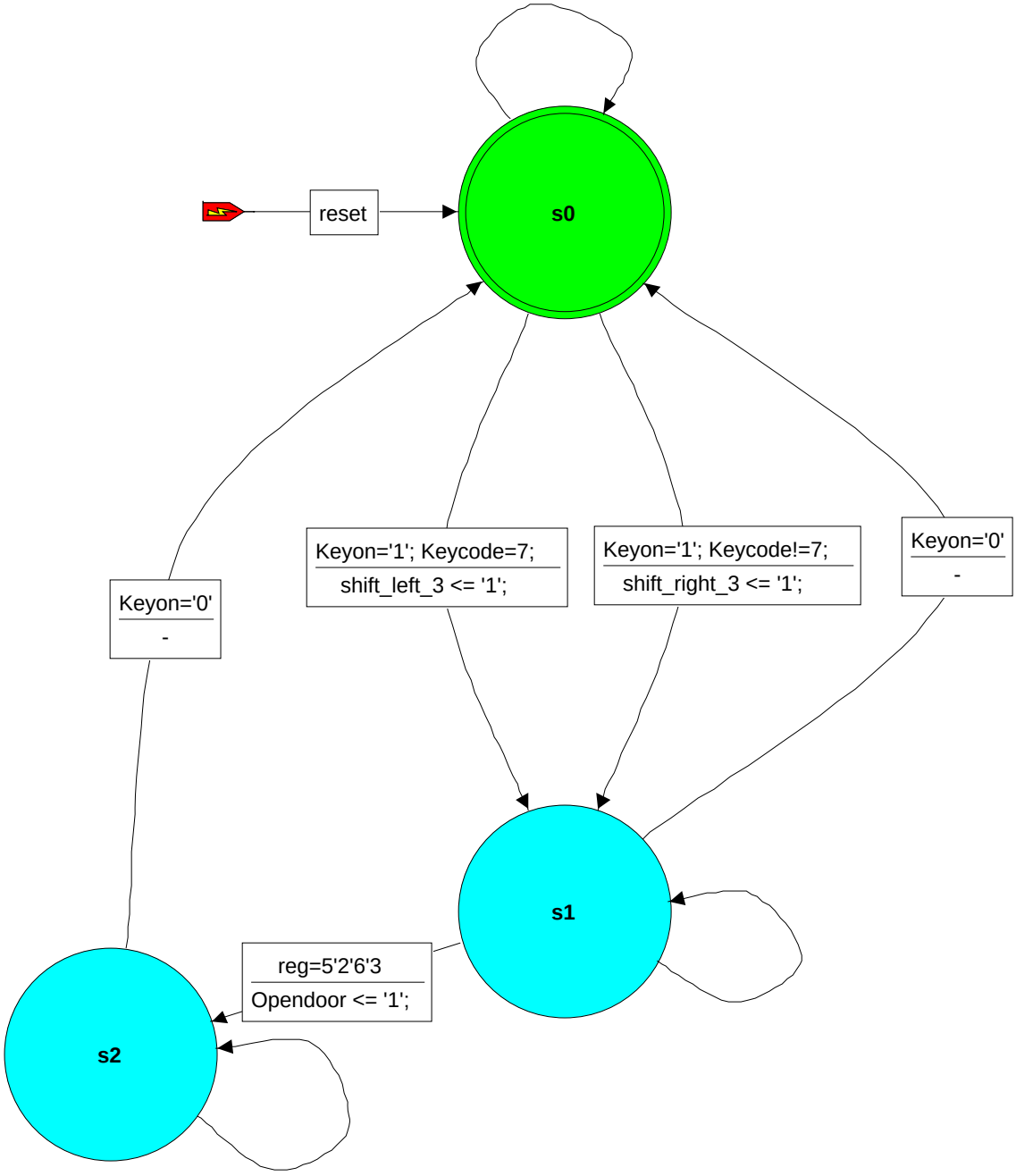
Count: 00 -> 00 -> 01 -> 10 -> 11 ->

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.


Keycode
clk
In
Shift_Reg
shift_right_3
shift_left_3
12
Keyon
Controler
Opendoor


reset
s0
Keyon='1'; Keycode=7;
shift_left_3 <= '1';
Keyon='1'; Keycode!=7;
shift_right_3 <= '1';
Keyon='0'
-
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;
```

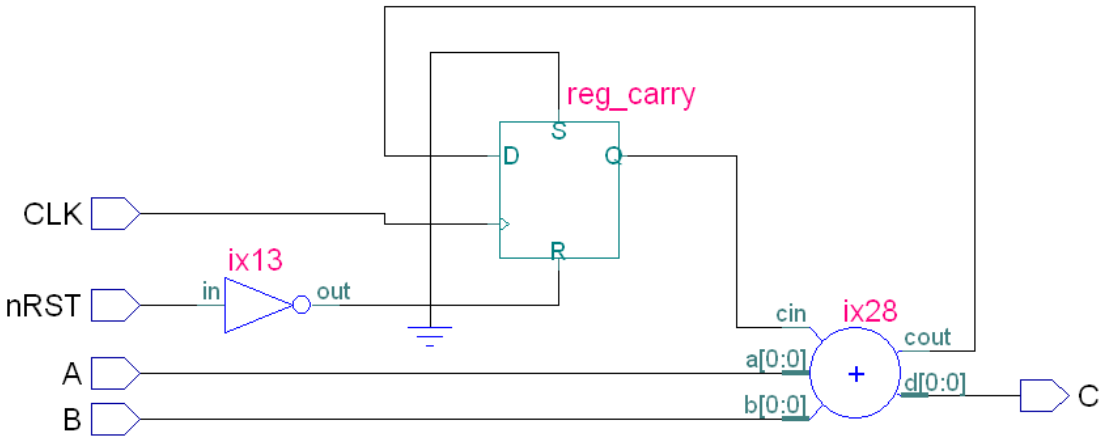
1. Dessinez le schéma logique correspondant.

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.

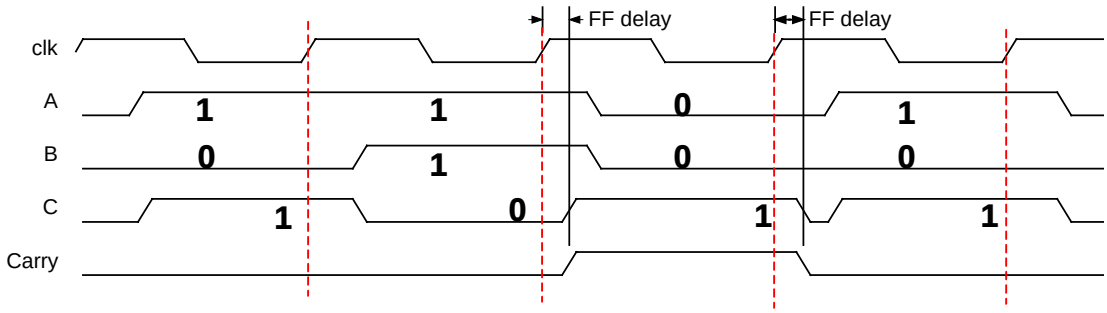
3. Décrivez en une phrase la fonction du circuit. Expliquez quel rôle a dans l'algorithme le composant réalisé par les lignes:

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

1.



2.

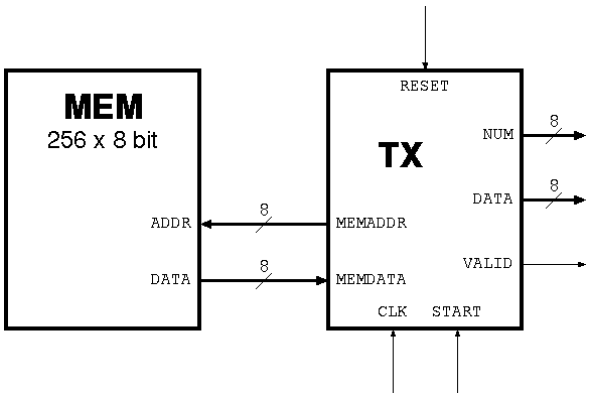


3.

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 256^{e} 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.

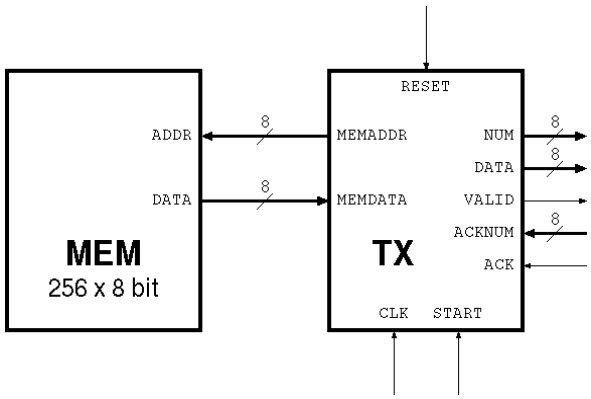
b. Ecrivez le code VHDL qui réalise le transmetteur.

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 :

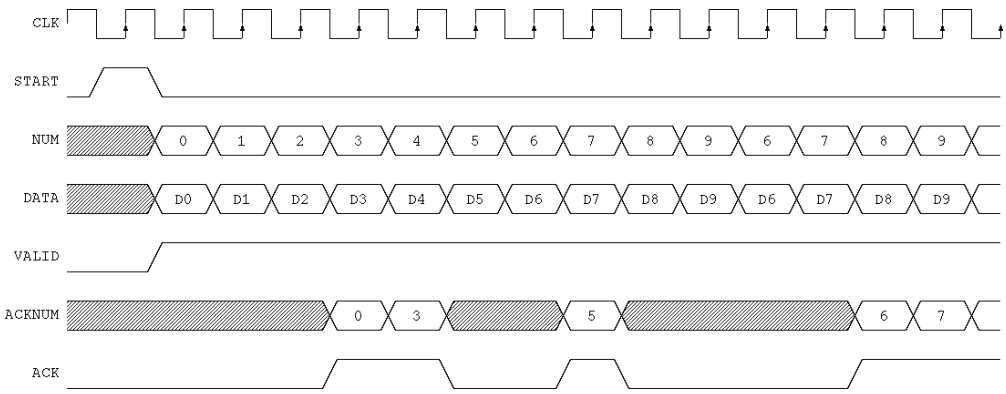
   - Le transmetteur reçoit des acquittements de la part du récepteur lorsque le signal ACK est actif. Si ACK est actif, le transmetteur peut lire le numéro du mot acquitté sur le bus ACKNUM. Ces acquittements informent le transmetteur du dernier mot bien reçu à destination. Remarquez que le récepteur n'est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

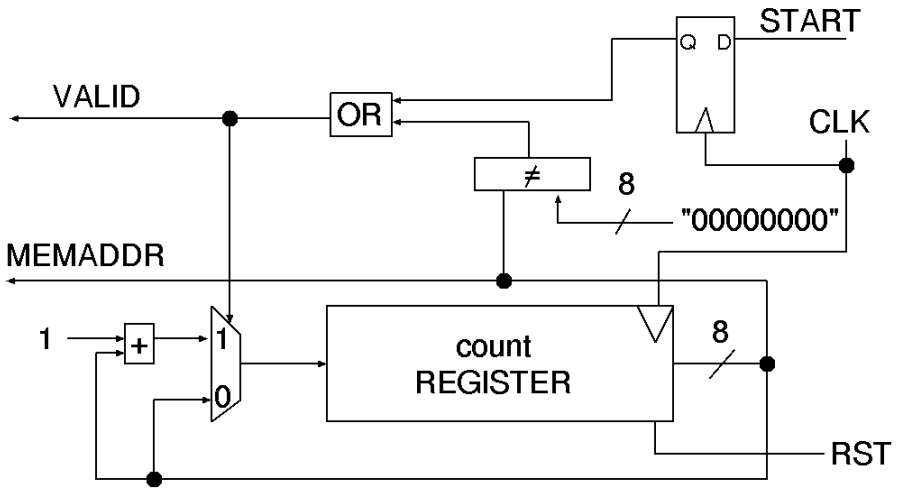
Modifiez le schéma à blocs du transmetteur TX pour réaliser ce protocole. Ignorez les détails du protocole à la fin de

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.

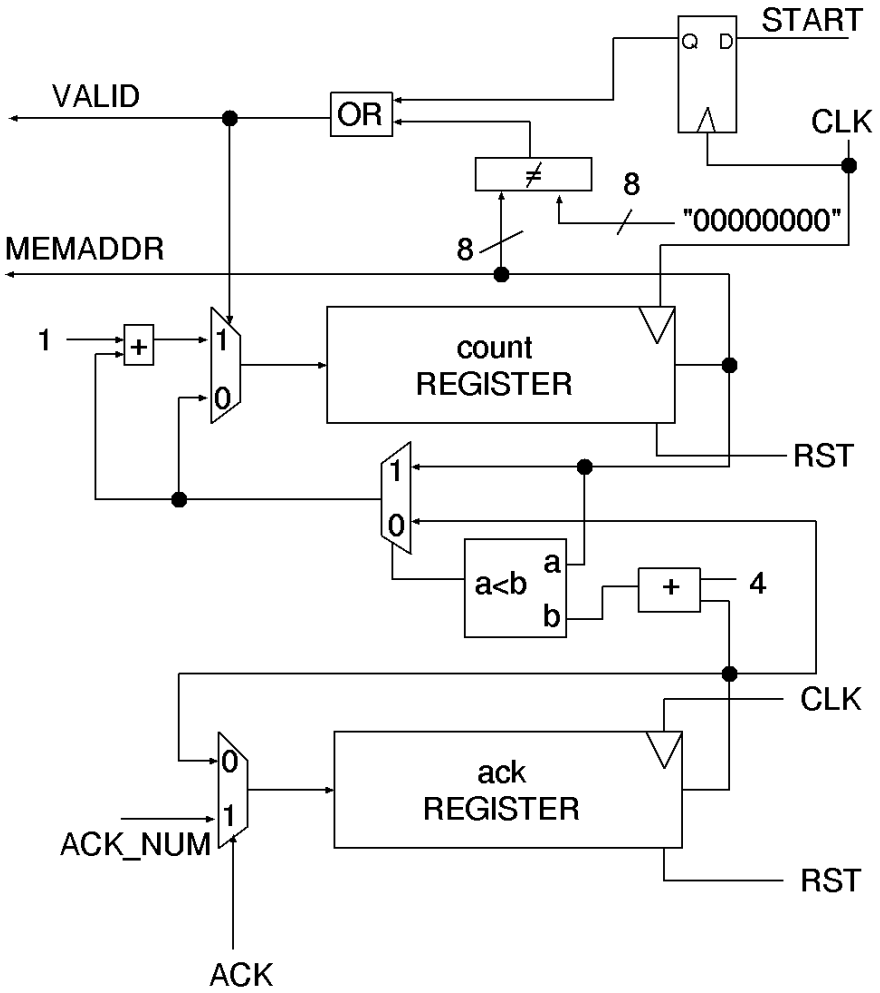


b)

```
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;
```

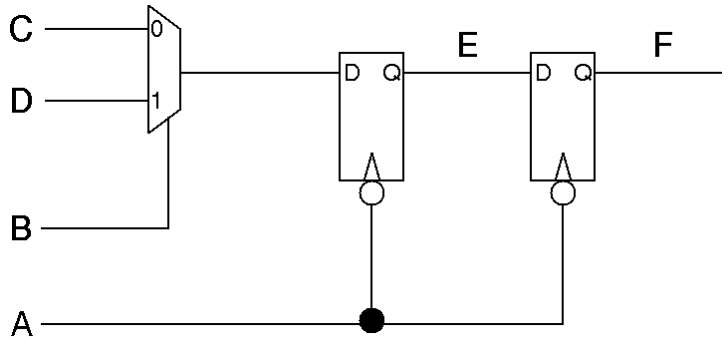
a. Dessinez le circuit correspondant si le signal E était remplacé par une variable.

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 :



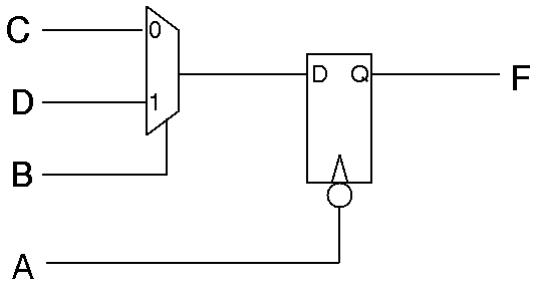
b)
Si l'on remplace E par une variable, comme suit :

```
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;
```

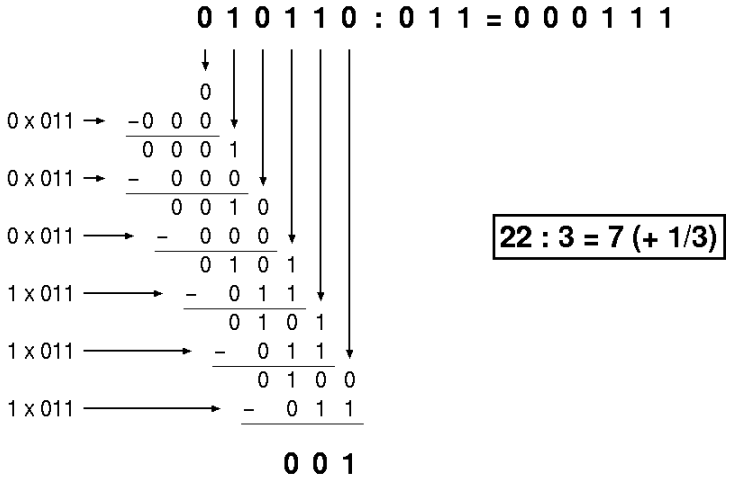
ce qui correspond au circuit :



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 :



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).

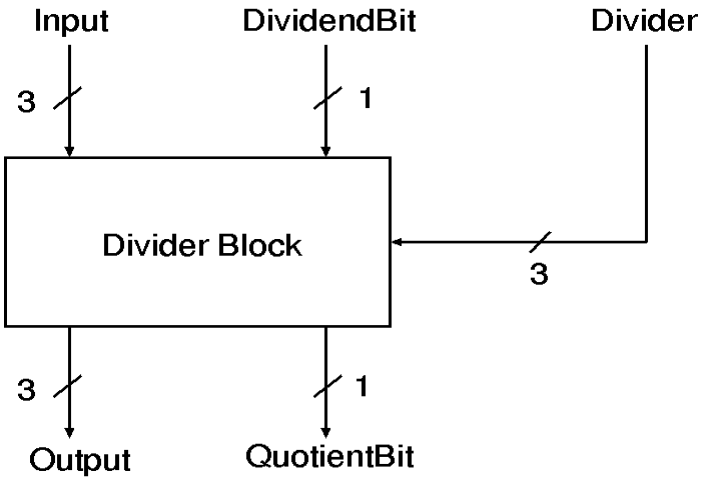
a. Décrivez les signaux d'entrée et de sortie du circuit qui réalise une itération de l'algorithme de division. Spécifiez le nombre de bits nécessaires pour encoder chacun des signaux et expliquez clairement la précision requise. (Supposez que toutes les copies de ce circuit sont identiques, même si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

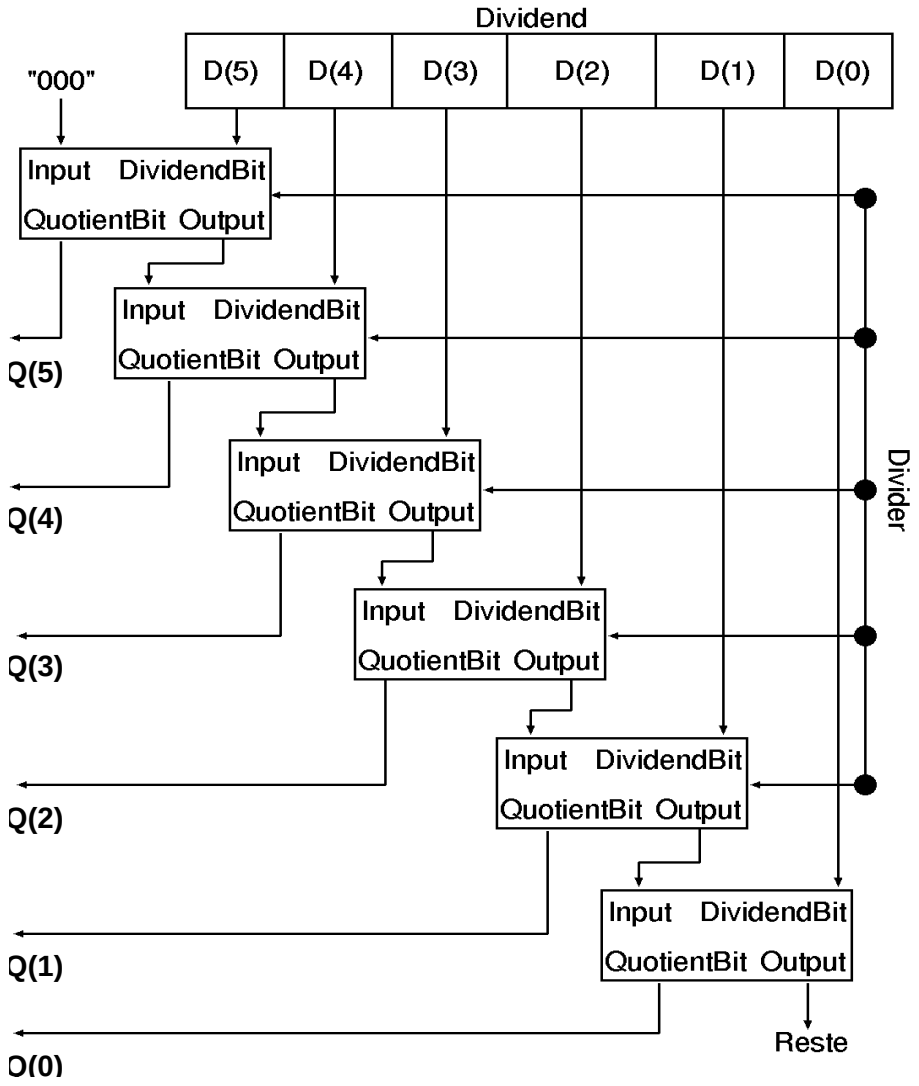
Commentaires :

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.

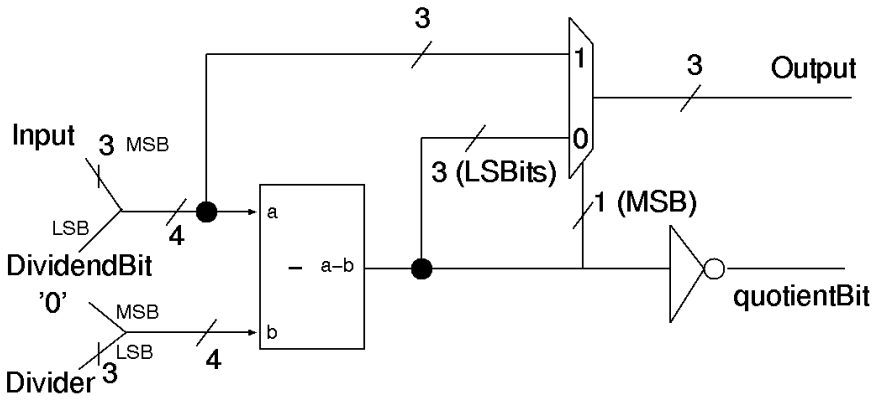
2. Le circuit a les sorties :
   - QuotientBit—le bit du quotient qui est obtenue à l'itération en question, et,
   - Output—3 bits de reste utilisés lors de l'itération suivante, ou, pour obtenir le reste final, lors de la dernière itération.

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.

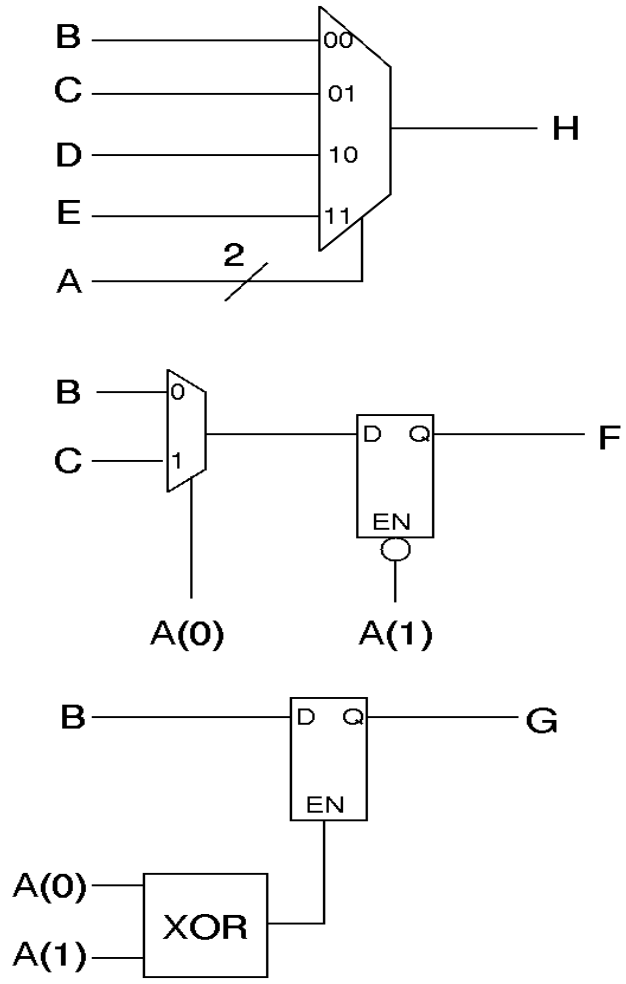
```
process(Input, DividendBit, Divider)
 signal sub : std_logic_vector(3 downto 0);
begin

 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 :



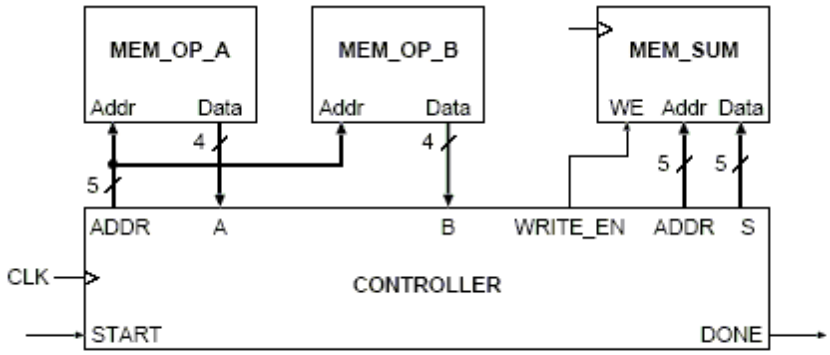
b)
Le signal H est issu d'un multiplexeur a 4 entrées.
Le signal F est mis à jour seulement lorsque A(1) vaut 0, et, prend alors une valeur dépendent de A(0). Quant au signal G, il est seulement mis a jour lorsque A(0) et A(1) sont différents.

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;
```

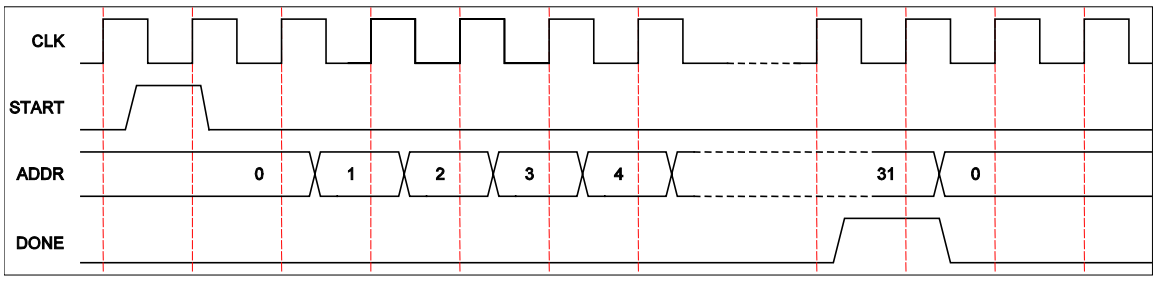
Considérez le système décrit dans la figure suivante :



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?

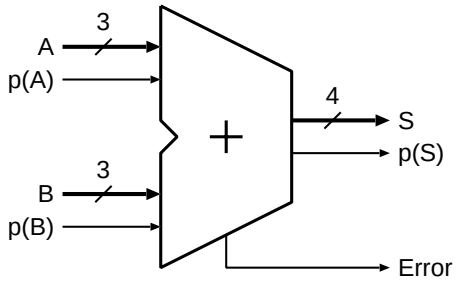
b. Décrivez le fonctionnement du contrôleur par une machine à états finis. Dessinez le diagramme des états. S'agit-il d'une machine de Mealy ou Moore ?

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.

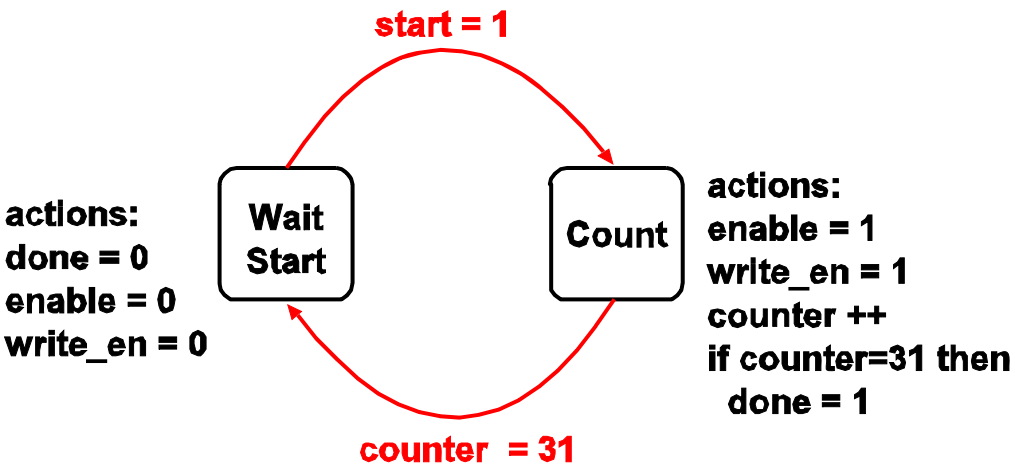
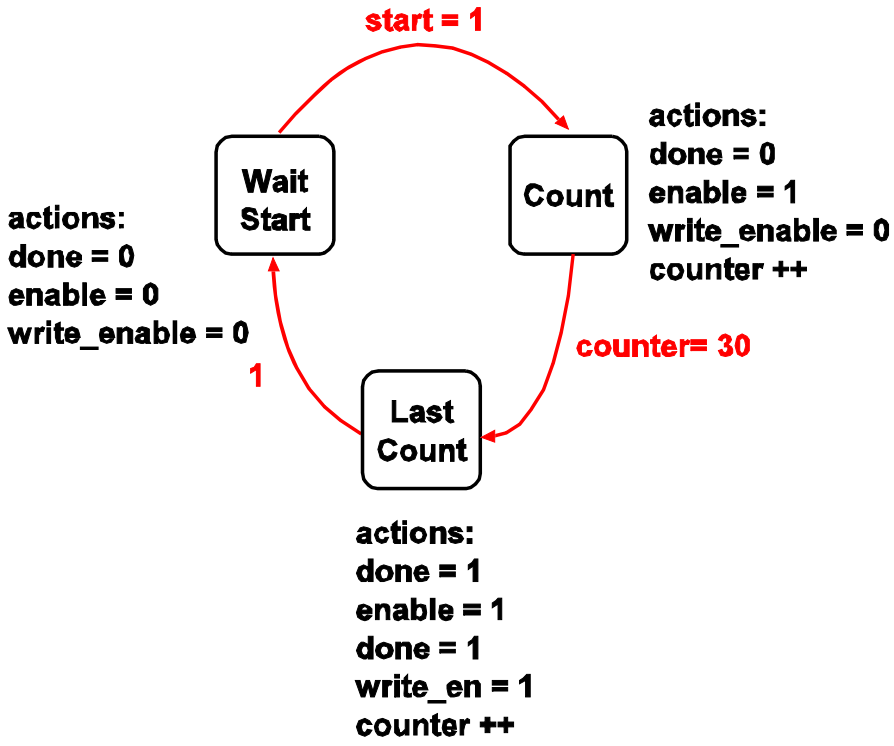


Figure 1

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

c)

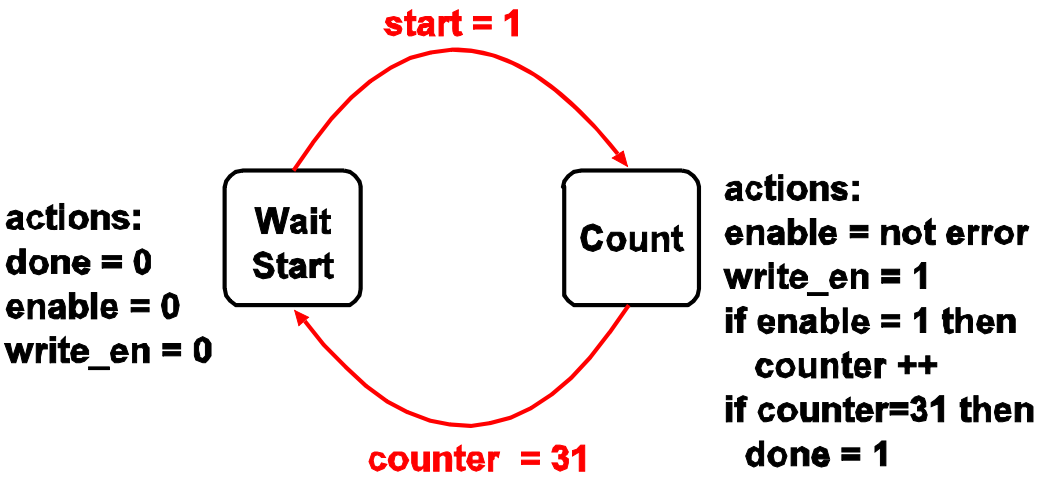
Le VHDL décrivant le système complet avec une machine d'état comme celle de la figure 1 est :

```
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 :

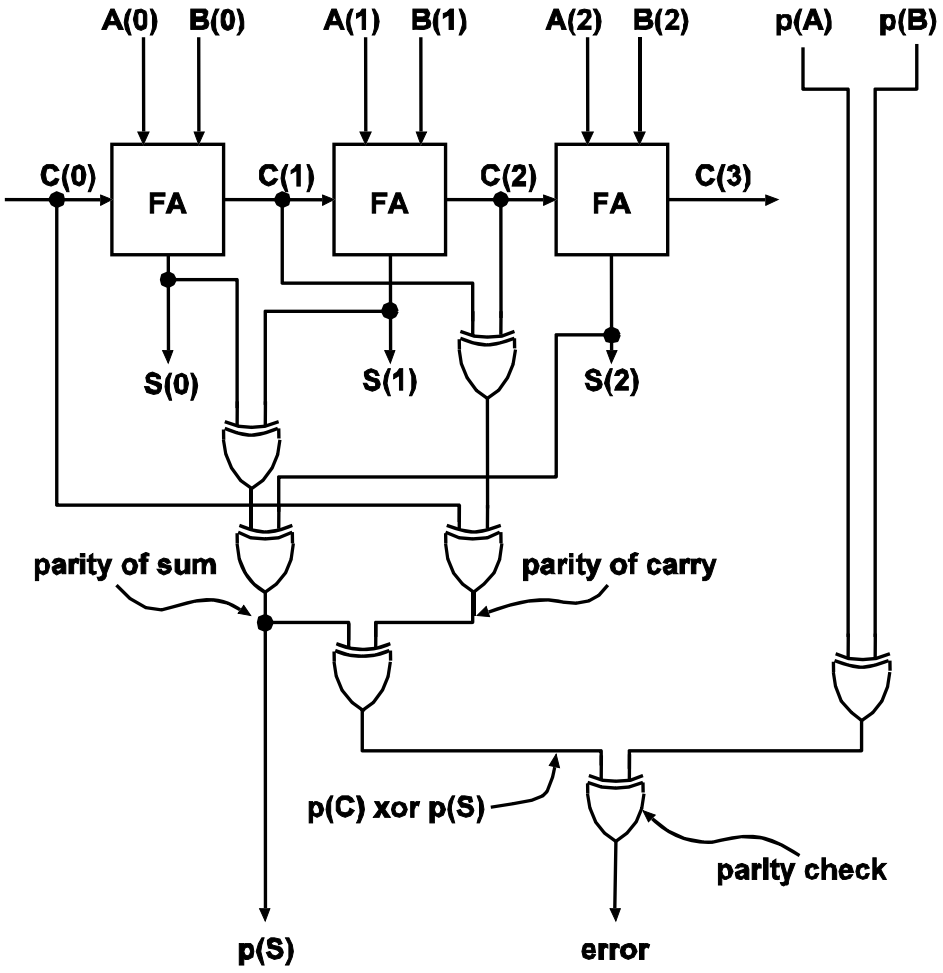


Le VDHL correspondant est identique à celui de la question c), sauf :

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;
```

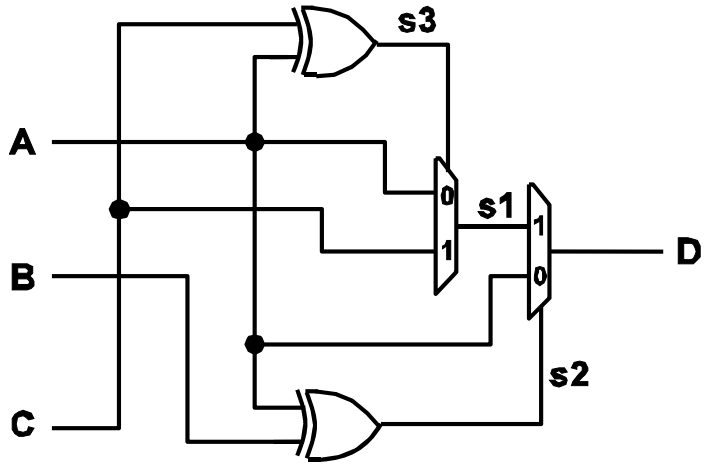
a. Dessinez le circuit décrit par ce code. Est-il séquentiel ou combinatoire? Justifiez votre réponse de façon concise.

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 :

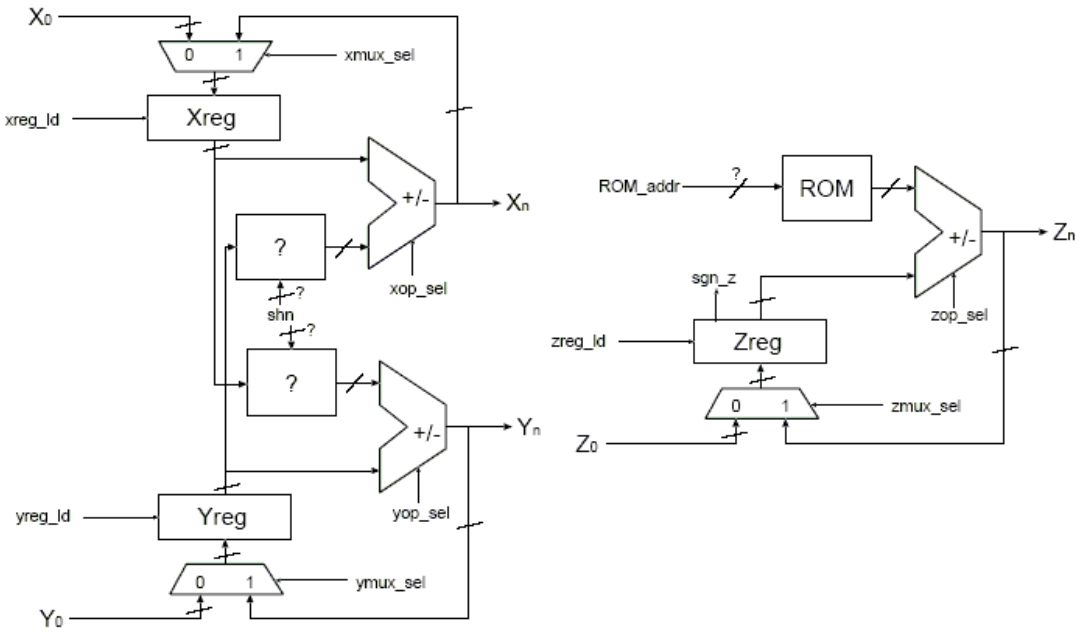
$$x_{i+1} = x_i - d_i \cdot y_i \cdot 2^{-i},$$
$$y_{i+1} = y_i + d_i \cdot x_i \cdot 2^{-i},$$
$$z_{i+1} = z_i - d_i \cdot \tan^{-1}(2^{-i}),$$

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 :

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

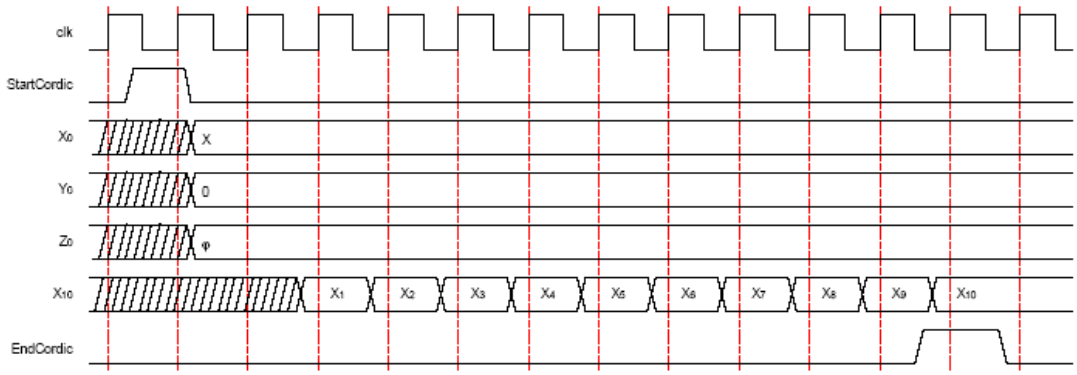
On utilise ces formules pour calculer les fonctions trigonométriques d'un angle φ. Le schéma du système qui réalise cette fonctionnalité est donné dans la figure suivante :



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.

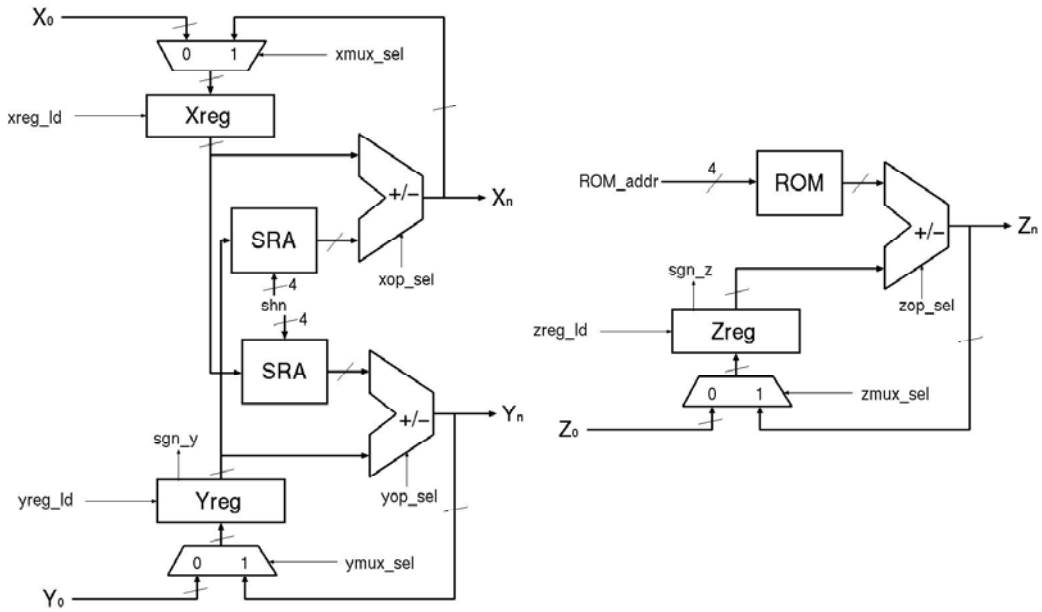
d. Ecrivez le code VHDL du contrôleur complet.

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.

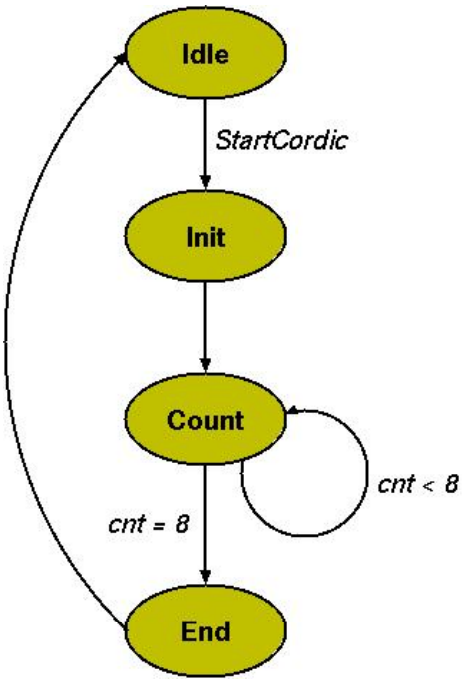



```
entity SRA is
  port(
    XY : in std_logic_vector(10 downto 0);
    shn : in std_logic_vector(3 downto 0);
    S: out std_logic_vector (10 downto 0);
  )
end entity SRA;
```

```
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.



Idle: all outputs '0'

Init: xreg_ld, yreg_ld, zreg_ld <– '1',
reset_cnt <– '1',
all other outputs '0'

Count: xreg_ld, yreg_ld, zreg_ld <– '1',
xmux_sel, ymux_sel, zmux_sel <– '1',
shn, ROM_addr <– cnt,
increase_cnt <– '1',
all other outputs '0'

End: EndCordic <–'1',
xmux_sel, ymux_sel, zmux_sel <– '1',
shn, ROM_addr <– cnt,
all other outputs '0'

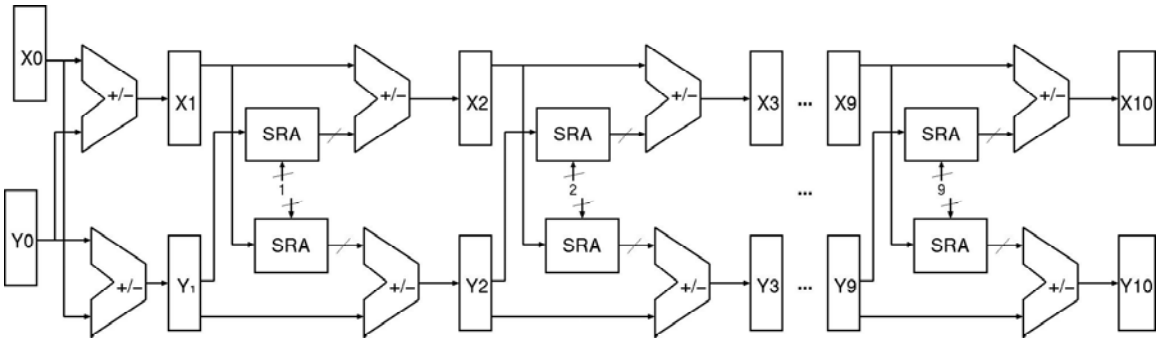
d)
Le code VHDL du contrôleur est donné ci-dessous :

```
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.

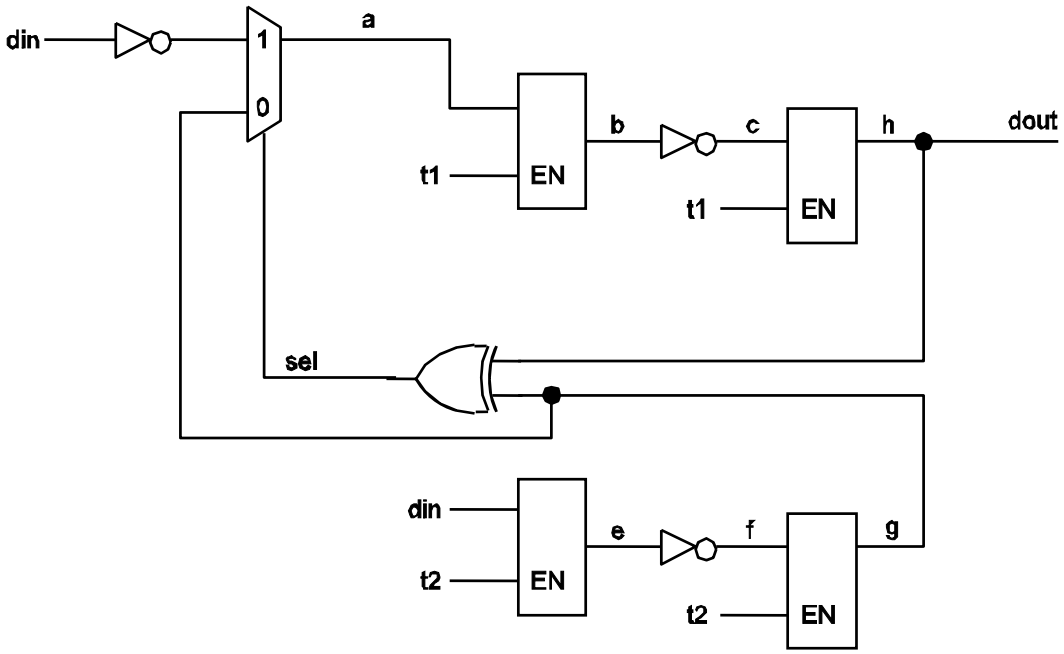

X0
Y0
+/−
X1
Y1
SRA
1
X2
Y2
2
X3
Y3
X9
Y9
9
X10
Y10

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)

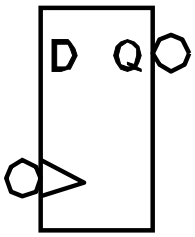


b)

*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.

Consider the MIPS program that follows (if necessary, use Figure 4.43 on page 274 of COD in order to recall of the MIPS syntax)

```
start: add  $v0, $zero, $zero
       add  $t0, $zero, $zero
outer: sltu $t2, $t0, $a1
       beq  $t2, $zero, fin
       lw   $t3, 0($a0)
       addi $t4, $zero, 32
inner: beq  $t4, $zero, next
       andi $t1, $t3, 1
       add  $v0, $v0, $t1
       srl  $t3, $t3, 1
       subi $t4, $t4, 1
       j    inner
next:  addi $t0, $t0, 1
       addi $a0, $a0, 4
       j    outer
fin:   jr   $ra
```

1. Describe the function of the program in one sentence.
2. Is it necessary to use the two instructions sltu/beq for the loop test, and is it possible to use one instruction instead? If possible, simplify the program.
3. Mark the instruction(s) that can cause an overflow (ignoring the instructions that compute addresses and indices).
4. Correct the program in order to take into account a possible overflow and return –1 in the place of the result if the overflow occurs (again, ignore the instructions that compute addresses and indices).
5. Is it possible to modify the program and minimize the number of times the internal loop is being executed? Show the modifications of the program (an idea: the loop counter is not necessary).

Solution 1:

The program counts the cumulative number of bits equal to one in all 32-bit words of an array.

Input:
$a0 – address of an array (N 32-bit elements)
$a1 – number of elements, N
Output:
$v0 – result

The code with comment:

```
start: add  $v0, $zero, $zero   # v0 = 0
       add  $t0, $zero, $zero   # t0 = 0
outer: sltu $t2, $t0, $a1       # t2 = (t0 < a1)
       beq  $t2, $zero, fin     # if (!t2) goto fin
       lw   $t3, 0($a0)         # t3 = mem[a0]
       addi $t4, $zero, 32      # t4 = 32
inner: beq  $t4, $zero, next    # if (!t4) goto next
       andi $t1, $t3, 1         # t1 = t3 & 1
       add  $v0, $v0, $t1       # v0 = v0 + t1
       srl  $t3, $t3, 1         # t3 = t3 >> 1
       subi $t4, $t4, 1         # t4 = t4 - 1
       j    inner               # goto inner
next:  addi $t0, $t0, 1         # t0 = t0 + 1
       addi $a0, $a0, 4         # a0 = a0 + 4
       j    outer               # goto outer
fin:   jr   $ra                 # return to caller
```

Solution 2:

It is not necessary to use both sltu/beq intructions. $t0 is initialised to 0 and is an incrementing index. It is sufficient to compare when it is equal to the number of elements rather than comparing when it is equal or larger

if ($t0 >= $a1) goto... => if ($t0 == $a1) goto...

The code Without sltu:

```
start: add  $v0, $zero, $zero
       add  $t0, $zero, $zero
outer: beq  $t0, $a1, fin       # if (t0==a1) goto fin
       lw   $t3, 0($a0)
       addi $t4, $zero, 32
inner: beq  $t4, $zero, next
       andi $t1, $t3, 1
       add  $v0, $v0, $t1
       srl  $t3, $t3, 1
       subi $t4, $t4, 1
       j    inner
next:  addi $t0, $t0, 1
       addi $a0, $a0, 4
       j    outer
fin:   jr   $ra
```

Solution 3:

If we ignore the instructions that compute addresses and indices, an overflow may be caused by the only arithmetic instruction in the loop:

add $v0, $v0, $t1

Eventual Overflow Cause:

```
start: add  $v0, $zero, $zero
       add  $t0, $zero, $zero
outer: sltu $t2, $t0, $a1
       beq  $t2, $zero, fin
       lw   $t3, 0($a0)
       addi $t4, $zero, 32
inner: beq  $t4, $zero, next
       andi $t1, $t3, 1
       add  $v0, $v0, $t1       # < - - - - - -
       srl  $t3, $t3, 1
       subi $t4, $t4, 1
       j    inner
next:  addi $t0, $t0, 1
       addi $a0, $a0, 4
       j    outer
fin:   jr   $ra
```

Solution 4:

One way to detect an addition overflow is to check if the result sign is not as expected.
Another way to detect an overflow (in this specific context) is to realize that the overflow is possible only when $v0 = 0x7FFFFFFF, and $t1 = 1

The Code With Overflow Handling (I):

```
start: add  $v0, $zero, $zero
       add  $t0, $zero, $zero
outer: sltu $t2, $t0, $a1
       beq  $t2, $zero, fin
       lw   $t3, 0($a0)
       addi $t4, $zero, 32
inner: beq  $t4, $zero, next
       andi $t1, $t3, 1
       add  $v0, $v0, $t1
       slt  $t5, $v0, $zero     # set t5 if v0 becomes
                                # negative
       bne  $t5, $zero, error   # if (t5) goto error
       srl  $t3, $t3, 1
       subi $t4, $t4, 1
       j    inner
next:  addi $t0, $t0, 1
       addi $a0, $a0, 4
       j    outer
error: addi $v0, $zero, -1      # v0 = -1 (overflow)
fin:   jr   $ra
```

The Code With Overflow Handling (II):

```
start: add  $v0, $zero, $zero
       add  $t0, $zero, $zero
outer: sltu $t2, $t0, $a1
       beq  $t2, $zero, fin
       lw   $t3, 0($a0)
       addi $t4, $zero, 32
inner: beq  $t4, $zero, next
       andi $t1, $t3, 1
       add  $v0, $v0, $t1
       srl  $t5, $v0, 31        # get the sign of v0
                                # into t5
       bne  $t5, $zero, error   # if (t5) goto error
       srl  $t3, $t3, 1
       subi $t4, $t4, 1
       j    inner

next:  addi $t0, $t0, 1
       addi $a0, $a0, 4
       j    outer

error: addi $v0, $zero, -1      # v0 = -1 (overflow)

fin:   jr   $ra
```

The Code with Overflow Handling (III):

```
start: add  $v0, $zero, $zero
       add  $t0, $zero, $zero
       addi $t5, $zero, 1       # t5 = 1
       sll  $t5, $t5, 31        # t5 = 0x80000000

outer: sltu $t2, $t0, $a1
       beq  $t2, $zero, fin
       lw   $t3, 0($a0)
       addi $t4, $zero, 32
inner: beq  $t4, $zero, next
       andi $t1, $t3, 1
       add  $v0, $v0, $t1
       beq  $v0, $t5, error     # if (v0==0x80000000)
                                # goto error
       srl  $t3, $t3, 1
       subi $t4, $t4, 1
       j    inner
next:  addi $t0, $t0, 1
       addi $a0, $a0, 4
       j    outer
error: addi $v0, $zero, -1      # v0 = -1 (overflow)

fin:   jr   $ra
```

Solution 5:

Idea: Instead of counting the number of inner loop iterations one can just test if the word under examination becomes null. In that case, it is useless to continue the inner loop, since it will make no contribution to the result.
The Code Without Inner Counter:

```
start: add  $v0, $zero, $zero
       add  $t0, $zero, $zero
outer: sltu $t2, $t0, $a1
       beq  $t2, $zero, fin
       lw   $t3, 0($a0)
inner: beq  $t3, $zero, next    # if (!t3) goto next
       andi $t1, $t3, 1
       add  $v0, $v0, $t1
       srl  $t3, $t3, 1
       j    inner
next:  addi $t0, $t0, 1
       addi $a0, $a0, 4
       j    outer
fin:   jr   $ra
```

MIPS program to study:

```
        add     $t0, $zero, $zero
        add     $v0, $zero, $zero
        add     $v1, $zero, $zero
Loop:   sltu    $t2, $t0, $a1
        beq     $t2, $zero, fin
        lw      $t1, 0($a0)
        sltu    $t2, $t1, $v0
        bne     $t2, $zero, skip
        add     $v0, $t1, $zero
        add     $v1, $t0, $zero
Skip:   Addi    $t0, $t0, 1
        addi    $a0, $a0, 4
        j       loop
fin:
```

QUESTIONS:

1. The program inputs are $a0, $a1. The program outputs are $v0, $v1. What does this program (in a sentence)? What are the values returned in $v0 and $v1?
2. What is type of quantities stored in the array (explain the answer)?
3. Adapt the program (1): make it a procedure (the values of $a0 and $a1 should be preserved).
4. Adapt the program (2): keep its functionality but let it operate on an array of unsigned bytes (avoid using the lb instruction!). Assume MIPS is a little-endian machine.
5. Adapt the program resulting from question 4 considering that the words in memory have been stored previously in big-endian.

Solution 1:

Find Maximum ($v0) and Index ($v1)

```
max:    add     $t0, $zero, $zero   # t0 = 0
        add     $v0, $zero, $zero   # v0 = 0
        add     $v1, $zero, $zero   # v1 = 0
loop:   sltu    $t2, $t0, $a1       # t2 = (t0 < a1)
        beq     $t2, $zero, fin     # if (!t2) goto fin
        lw      $t1, 0($a0)         # t1 = mem[a0]
        sltu    $t2, $t1, $v0       # t2 = (t1 < v0) UNSIGNED!
        bne     $t2, $zero, skip    # if (t2) goto skip
        add     $v0, $t1, $zero     # v0 = t1
        add     $v1, $t0, $zero     # v1 = t0
skip:   addi    $t0, $t0, 1         # t0 = t0 + 1
        addi    $a0, $a0, 4         # a0 = a0 + 4
        j       loop                # goto loop
fin:                                # finish
```

Solution 2:

-

Solution 3:

Procedure (an easy way)

```
max:    Add     $t3, $a0, $zero     # t3 = a0
        add     $t0, $zero, $zero
        add     $v0, $zero, $zero
        add     $v1, $zero, $zero
loop:   Sltu    $t2, $t0, $a1
        beq     $t2, $zero, fin
        lw      $t1, 0($t3)         # use t3 instead
        sltu    $t2, $t1, $v0
        bne     $t2, $zero, skip
        add     $v0, $t1, $zero
        add     $v1, $t0, $zero
skip:   addi    $t0, $t0, 1
        addi    $t3, $t3, 4         # use t3 instead
        j       loop
fin:    jr      $ra                 # return to caller
```

Procedure (using the stack)

```
max:    Subi   $sp, $sp, 8          # sp = sp – 8
        Sw     $ra, 4($sp)          # mem[sp + 4] = ra
                                    (optional)
        sw     $a0, 0($sp)          # mem[sp] = a0
        add    $t0, $zero, $zero
        add    $v0, $zero, $zero
        add    $v1, $zero, $zero
loop:   sltu   $t2, $t0, $a1
        beq    $t2, $zero, fin
        lw     $t1, 0($a0)
        sltu   $t2, $t1, $v0
        bne    $t2, $zero, skip
        add    $v0, $t1, $zero
        add    $v1, $t0, $zero
skip:   addi   $t0, $t0, 1
        addi   $a0, $a0, 4
        j      loop
fin:    lw     $a0, 0($sp)          # a0 = mem[sp]
        lw     $ra, 4($sp)          # ra = mem[sp+4] (optional)
        addi   $sp, $sp, 8          # sp = sp + 8
        jr     $ra                  # return to caller
```

Solution 4

Reminder: Big Endian and Little Endian

When storing a multi-byte word on several words, one has two choices: store the most or least significant part of the word in byte with the first (lowest) address:

```
sw 1234567816, (1000)
```

Address	Data
1000	78
1001	56
1002	34
1003	12

Little Endians
e.g., x86, DEC3100

Address	Data
1000	12
1001	34
1002	56
1003	78

Big Endians
e.g., 68k, PowerPC, Sparc

*Find Maximum in Byte Array (**lb**-way):*

```
max:    add   $t0, $zero, $zero
        add   $t1, $zero, $zero   # initialize
        add   $v0, $zero, $zero
        add   $v1, $zero, $zero
        sltu  $t2, $t0, $a1       # t2 = (t0 < a1)
        beq   $t2, $zero, fin     # if (!t2) goto fin

loop:   add   $t3, $a0, $t0       # prepare the next address
        lbu   $t1, 0($t3)         # load byte to t1
        sltu  $t2, $t1, $v0
        bne   $t2, $zero, skip
        add   $v0, $t1, $zero     # v0 = t1
        add   $v1, $t0, $zero     # v1 = t0

skip:   addi  $t0, $t0, 1         # if (t0==a1) goto fin
        beq   $t0, $a1, fin
        j     loop
fin:
```

*Well… the use of **lb** was disallowed…*

*Find Maximum in Byte Array (**lw**-way):*

```
max:    add   $t0, $zero, $zero
        add   $v0, $zero, $zero
        add   $v1, $zero, $zero
        sltu  $t2, $t0, $a1       # t2 = (t0 < a1)
        beq   $t2, $zero, fin     # if (!t2) goto fin
loop:   add   $t3, $a0, $t0       # prepare the next address
        lw    $t1, 0($t3)         # load word to t1
        andi  $t1, $t1, 0x00ff    # extract the LSB to T1
        sltu  $t2, $t1, $v0
        bne   $t2, $zero, skip
        add   $v0, $t1, $zero     # v0 = t1
        add   $v1, $t0, $zero     # v1 = t0
skip:   addi  $t0, $t0, 1
        beq   $t0, $a1, fin       # if (t0==a1) goto fin
        j     loop
fin:
```

Why is this suboptimal? (Count the memory accesses…)
***Note:** This program is actually wrong because the MIPS Architecture requires lw memory accesses to be word aligned—i.e., addresses must be multiple of 4.*

Find Maximum in Byte Array (best):

```
max:    add   $t0, $zero, $zero
        add   $v0, $zero, $zero
        add   $v1, $zero, $zero
        sltu  $t2, $t0, $a1       # t2 = (t0 < a1)
        beq   $t2, $zero, fin     # if (!t2) goto fin
outer:  lw    $t1, 0($a0)
inner:  andi  $t3, $t1, 0x00ff    # extract byte to t3
        sltu  $t2, $t3, $v0
        bne   $t2, $zero, skip
        add   $v0, $t3, $zero     # v0 = t3
        add   $v1, $t0, $zero
skip:   addi  $t0, $t0, 1
        beq   $t0, $a1, fin       # if (t0==a1) goto fin
        srl   $t1, $t1, 8         # prepare for the next
        andi  $t3, $t0, 0x0003    # word finished?
        bne   $t3, $zero, inner   # if not, goto inner
        addi  $a0, $a0, 4
        j     outer
fin:
```

Solution 5

```
max:    add   $t0, $zero, $zero
        add   $t4, $zero, $zero    # initialize t4 and
        lui   $t4, 0xff00          # prepare the mask
        add   $v0, $zero, $zero
        add   $v1, $zero, $zero
        sltu  $t3, $t0, $a1
        beq   $t3, $zero, fin
outer:  lw    $t1, 0($a0)
inner:  and   $t3, $t1, $t4
        sltu  $t2, $t3, $v0
        bne   $t2, $zero, skip
        add   $v0, $t3, $zero
        add   $v1, $t0, $zero
skip:   addi  $t0, $t0, 1
        beq   $t0, $a1, fin
        sll   $t1, $t1, 8
        andi  $t3, $t0, 0x0003
        bne   $t3, $zero, inner
        addi  $a0, $a0, 4
        j     outer

fin:    srl   $v0, 24              # align the result
```

Write a MIPS program that compares the sign of numbers in two arrays of 32-bit numbers and counts how many of them have a different sign.

The number's representation is 2s complement.

At the beginning, $a0 contains the memory address of the first array, $a1 contains the memory address of the second array, and $a2 the number of elements. The result should be returned in $v0. Besides $v0, only registers $t0-$t9 can be modified.

```
proc:  add   $v0, $zero, $zero   # initialize v0

loop:  beq   $a2, $zero, fin     # if (a2=0) goto fin
       lw    $t3, 0($a0)         # t3 = mem[a0]
       lw    $t4, 0($a1)         # t4 = mem[a1]
       xor   $t5, $t3, $t4       # t5 = t3 xor t4
       srl   $t5, $t5, 31        # t5 = t5 >> 31
       add   $v0, $v0, $t5       # v0 = v0 + t5

skip:  addi  $a2, $a2, -1        # a2 = a2 – 1
       addi  $a0, $a0, 4         # a0 = a0 + 4
       addi  $a1, $a1, 4         # a1 = a1 + 4
       j     loop                # goto loop

fin:
```

1. Ecrire le programme assembleur MIPS de la fonction *strlen(adr)*. Cette fonction compte le nombre le nombre de caractères (8-bits) d'une chaîne de caractères qui se termine par un caractère "null" (00), sans compter ce dernier. Considérer que le processeur est little-endian. L'argument de la fonction, qui est l'adresse du début de la chaîne de caractères en mémoire, est passé par le registre $a0. La fonction retourne le résultat dans $v0. Utiliser la pseudo-instruction suivante, qui charge le Byte de l'adresse $t0, dans $t1 avec extension du signe.

```
lb    $t1, 0($t0)     # t1(low byte)= mem[t0]
```

2. Considérer que le programme précédent est une procédure. Quelles sont les instructions à rajouter? Quelles instructions faudra-t-il rajouter pour que l'appel de cette procédure, par exemple depuis le programme principal, ne modifie le contenu d'aucun registre?

Solutions 1 et 2 :

```
fct:  addi   $sp, $sp, -4        # sp = sp – 4
      sw     $s0, 0($sp)         # mem[sp] = s0
      add    $s0, $zero, $zero   # s0 = 0

L1:   add    $t0, $a0, $s0       # t0 = a0 + s0
      lb     $t1, 0($t0)         # t1(low byte)= mem[t0]
      addi   $s0, $s0, 1         # s0 = s0 + 1
      bne    $t1, $zero, L1      # if (t1!=0) goto L1
      addi   $s0, $s0, -1        # s0 = s0 – 1
      add    $v0, $s0, $zero     # v0 = s0
      lw     $s0, 0($sp)         # s0 = mem[sp]
      addi   $sp, $sp, 4         # sp = sp + 4
      jr     $ra                 # goto ra
```

lb is not a basic MIPS instruction. We could replace it by the following equivalent code of lbu:

```
lbu:        andi      $t2, $t0, 0Xfffc
            lw        $t1, 0($t2)
            andi      $t3, $t0, 3

llbu:       beq       $t3, $zero, lbuend
            srl       $t1,      $t1, 8
            addi      $t3, $t3, -1
            j         llbu

lbuend:     andi      $t1, $t1, 0xFF
```

1. Ecrire un programme en assembleur MIPS capable d'effectuer la somme de deux nombres signés sur 32-bits, représentés en « *Sign-and-Magnitude* », stockés dans les registres **`$t0`** et **`$t1`**. Le résultat doit être placé dans **`$t2`**, également en format « *Sign-and-Magnitude* ». Utiliser exclusivement les registres **`$t0`** à **`$t7`**.

   Représentation « *Sign-and-Magnitude* » :

$$A =< sa_{n-2}...a_2a_1a_0 >= (-1)^s \cdot \sum_{i=0}^{n-2} a_i 2^i$$

   Indications sûr la méthode à suivre :

   a) Passer les deux opérandes du format « *Sign-and-Magnitude* » au format complément à deux.
   b) Additionner les nouveaux opérandes en complément à 2.
   c) Passer le résultat du format complément à 2 au format « *Sign-and-Magnitude* ».

   Dans un premier temps, les dépassements de capacité (*overflow*) ne sont pas traités.

   En plus des instructions de la figure 4.43 (page 274 de COD), vous pouvez utiliser également les instructions logiques suivantes :

```
xor     $s1, $s2, $s3    =>   $s1 = $s2 xor $s3
not     $s1, $s2         =>   $s1 = not $s2
```

2. Compléter le programme précédant de façon à ce que l'opération s'effectue avec des vecteurs de nombres (tableau de mots) comme opérandes. Le résultat sera placé dans un vecteur de nombres séparé.

   Considérer qu'au début du programme les paramètres d'entrée sont les suivants:

   `$a0` : Contient l'adresse du vecteur du 1er opérande.
   `$a1` : Contient l'adresse du vecteur du 2ème opérande.
   `$a2` : Contient le nombre d'éléments d'un vecteur (nombre de mots du tableau). Les vecteurs ont tous une taille identique.
   `$a3` : Contient l'adresse du vecteur du résultat.

3. Pour chacun des points a), b) et c) de la méthode indiquée dans la première question, indiquer s'il peut y avoir un dépassement de capacité ou un problème de conversion et décrire les façons possibles de les traiter.

Solution 1 :

```
        # Initialise $t5 avec une valeur qui sera utilisée
        comme masque
        ori     $t5, $zero, 0x8000      # { $t5=0x8000'0000
        sll     $t5, $t5, 16            # }

        # Test du signe:
        # Pour chacun des opérandes, on teste si la valeur
        # est positive et dans ce cas, on n'effectue pas la
        # conversion.
        srl     $t3, $t0, 31
        srl     $t4, $t1, 31
```

Autre méthode:

```
        slt     $t3, $t0, $zero
        slt     $t4, $t0, $zero
```

```
        beq     $t3, $zero, checkb      # if (t3=0) goto checkb

        # Conversion c. a 2 du premier opérande:
        xor     $t0, $t0, $t5           # Inverse bit de signe
        not     $t0, $t0                # t0 = not t0
        addiu   $t0, $t0, 1             # t0 = t0 + 1
```

Autres méthodes:

a)

```
        xor     $t0, $t0, $t5           # Inverse bit de signe
        sub     $t0, $zero, $t0         # t0 = 0 - t0
```

b)

```
        not     $t0, $t0                # t0 = not t0
        or      $t0, $t0, $t5
        addi    $t0, $t0, 1
```

```
checkb: beq     $t4, $zero, next        # if (t4=0) goto next

        # Conversion c. a 2 du deuxième opérande:
        xor     $t1, $t1, $t5           # Inverse bit de signe
        not     $t1, $t1                # t1 = not t1
        addiu   $t1, $t1, 1             # t1 = t1 + 1
next:   add     $t2, $t0, $t1           # Addition en c. a 2

        # Saute si le résultat est positif
        Srl     $t7, $t2, 31            # t7 = t2 >> 31
        Beq     $t7, $zero, fin         # if (t7 = 0) goto fin
```

Autre méthode:

```
        slt     $t7, $t2, $zero
        beq     $t7, $zero, fin
```

```
        # résultat: c. a 2 -> Sign-and-magnitude
        addi    $t2, $t2, -1
        not     $t2, $t2
        xor     $t2, $t2, $t5
fin:
```

Autre méthode:

```
        not     $t2, $t2
        addi    $t2, $t2, 1
        or      $t2, $t2, $t5
```

Solution 2 :

```
loop:   beq  $a2, $zero, out
        lw   $t0, 0($a0)
        lw   $t1, 0($a1)

        # ...

        sw   $t2, 0($a3)

        addi $a0, $a0, 4
        addi $a1, $a1, 4
        addi $a2, $a2, -1
        addi $a3, $a3, 4

        j    loop
out:
```

Solution 3 :

a) Conversion Sign-and-Magnitude -> complément à 2:
Sign-and-Magnitude : $-(2^{n-1}-1)$ *à* $2^{n-1}-1$
Complément à 2 : $-(2^{n-1})$ *à* $2^{n-1}-1$
⇨ *pas de problème de conversion*

b) Overflow d'un cas normal d'addition de 2 valeurs signées en complément à 2 :
- *si les signes des opérandes sont différents => pas d'Overflow*
- *si les dignes des opérandes sont identiques => Overflow lorsque le signe du résultat est différent des opérandes*

c) $-(2^{n-1})$ *ne peut pas être représenter en Sign-and-Magnitude.*

1. Les instructions suivantes, en assembleur MIPS, représentent une structure de contrôle très courante dans les langages de programmation de haut niveau (Java, Ada ou C, par exemple) : quelle est cette structure ?

```
        slt         $t0, $a0, $a1
        bne         $t0, $zero, cout
        … série d'instructions…
cout :
```

2. Ecrire des programmes en assembleur MIPS équivalents aux pseudo-instructions suivantes. Si nécessaire, utiliser **$t0** pour mémoriser des valeurs intermédiaires. Aucun autre registre ne peut être utilisé.

   a) add ($s0),$s1,($s2) => mem[$s0]=$s1+mem[$s2]

   Cette instruction MIPS n'existe pas, car elle utilise un mode d'adressage qui n'est pas supporté par les processeurs RISC.

   b) SWAP $s0 => bits 31-16 ↔ bits 15-0

   Cette instruction permet d'échanger les 16-bits de poids fort avec les 16-bits de poids faible.

   c) PUSH $s0

   Cette instruction n'est pas non plus une instruction MIPS. Elle décrémente le pointeur de pile (**SP**), puis sauve **$s0** à cette adresse.

3. Décoder les deux instructions MIPS suivantes :

Adresse:	Code:
0x10000000	**0x20080020**
0x10000004	**0x8D090004**

Solution 1 :

if ($a0 >= $a1) {
… série d'instructions …
}

Solution 2 :

add ($s0), $s1, ($s2) :

```
lw   $t0, 0($s2)
add  $t0, $s1, $t0
sw   $t0, 0($s0)
```

swap $s0 :

```
sll  $t0, $s0, 16
srl  $s0, $s0, 16
add  $s0, $s0, $t0
```

push $s0:

```
addi $sp, $sp, -4
sw   $s0, 0($sp)
```

Solution 3 :

0x20080020 => `addi $t0, $zero, 0x20`
0x8D090004 => `lw $t1, 4($t0)`

Analysez le programme suivant, en supposant qu'au début de son exécution les registres du processeur contiennent les informations suivantes:

$a0 : l'adresse d'une matrice de nombres non signés 8 bits
$a1 : le nombre de lignes de la matrice
$a2 : le nombre de colonnes de la matrice
$a3 : une valeur 8 bits non signée

```
start: add     $t0, $zero, $zero
       add     $t1, $zero, $zero
       add     $t3, $zero, $zero
loop:  lbu     $t2, 0($a0)
       add     $t2, $t2, $a3
       slt     $t4, $t2, $t3
       bne     $t4, $zero, skip
       add     $v0, $t0, $zero
       add     $v1, $t1, $zero
       add     $t3, $t2, $zero
skip:  sb      $t2, 0($a0)
       addi    $a0, $a0, 1
       addi    $t0, $t0, 1
       bne     $t0, $a1, loop
       add     $t0, $zero, $zero
       addi    $t1, $t1, 1
       bne     $t1, $a2, loop
end:
```

a. Décrivez en une phrase la fonction du programme (en supposant qu'il n'y a aucun dépassement de capacité). En particulier, donnez le contenu des registres **$v0** et **$v1** à la fin de son exécution, si **$a3**=0 et la matrice est:

$$\begin{pmatrix} 12 & 34 & 56 \\ 78 & 113 & 24 \\ 35 & 46 & 57 \\ 11 & 122 & 33 \end{pmatrix}$$

b. Donnez le contenu des adresses de mémoire 1000 à 1008, si la matrice précédente est stockée à partir de l'adresse 1000

c. Est-ce que l'instruction

```
add  $t2, $t2, $a3
```

du programme précédent peut générer un résultat qui n'est pas représentable sur 32 bits? Est-ce qu'elle peut générer un résultat non représentable sur 8 bits? Abandonnez l'hypothèse concernant l'absence de dépassement de capacité et modifiez le programme pour saturer le résultat en cas de dépassement (si le résultat n'est pas représentable, remplacez-le par la plus grande valeur possible).

a. Ce programme donne les coordonnées du nombre le plus grand d'une matrice. $v0 retourne l'indice de la ligne et $v1 l'indice de la colonne. Le contenu de la matrice est mis à jour avec la somme du contenu initial de la matrice et de la valeur de $a3. Pour l'exemple donné ($a3=0), le programme retourne $v0=3 et $v1=1 et le contenu de la matrice ne change pas.

b.

Adresse:	Donnée:
1000	12
1001	78
1002	35
1003	11
1004	34
1005	113
1006	46
1007	122
1008	56

c.

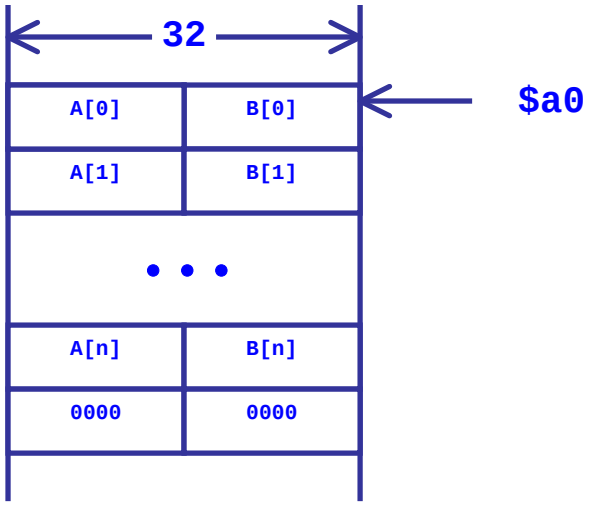
1. *La valeur max. de $t2 est 0xFF (8-bits non signé) et celle de $a3 est 0xFF (8-bits non signé). La somme est donc toujours représentable sur 9-bits, donc représentable sur 32-bits.*
2. *La somme n'est pas toujours représentable sur 8-bits.*
3. *Comme le résultat doit tenir sur 8-bits, il peut y avoir un dépassement de capacité sur l'instruction suivante:*

   add $t2, $t2, $a3

   Un dépassement de capacité est détecté après cette instruction en apportant la modification suivante au programme:

```
        …
        add  $t2, $t2, $a3

        srl  $t5, $t2, 8
        beq  $t5, $zero, no_V
        addi $t2, $zero, 0x00FF
no_V:
        slt  $t4, $t2, $t3
        …
```

Supposez que deux vecteurs **A** et **B**, des valeurs signées sur 16 bits, sont stockés en mémoire, à partir d'une adresse stockée dans le registre **$a0**. Les valeurs **A[i]** et **B[i]** sont stockées dans le même mot 32 bits de la mémoire (occupant de ce fait 4 adresses différentes). La fin des vecteurs est donnée par un mot égal à zéro.
Ecrivez un programme calculant le nombre de fois que **A[i]>B[i]**. Cette valeur doit se trouver dans **$v0** à la fin du programme.



```
proc:     addi                $v0, $zero, 0     #v0 = 0

loop:     lw                  $t2, 0($a0)
    beq   $t2, $zero, finish

    sra   $t3, $t2, 16        #t3 = (Sign Ext) & A[n]

    sll   $t2, $t2, 16        #t2 = B[n] & 0x0000
    sra   $t2, $t2, 16        #t2 = (Sign Ext) & B[n]

    slt   $t4, $t2, $t3       #if(t2<t3) t4=1 else t4=0
    addi  $v0, $v0, $t4

    addi  $a0, $a0, 4
    j     loop

finish:
```

Analysez le programme MIPS suivant, en supposant qu'au début de son exécution:

- le registre **$a0** contient l'adresse en mémoire d'une séquence de caractères codés en ASCII;
- la séquence de caractères se termine par un byte à zéro.

```
prog:     add  $v0, $zero, $zero
          addi $t0, $zero, 32
          addi $t1, $zero, 1

loop:     lbu  $t2, 0($a0)
          beq  $t2, $zero, finish
          beq  $t2, $t0, space
          beq  $t1, $zero, next
          addi $v0, $v0, 1
          add  $t1, $zero, $zero
          j    next

space:    addi $t1, $zero, 1
next:     addi $a0, $a0, 1
          j    loop
finish:
```

Rappelez-vous qu'un caractère espace (" ") est représenté en ASCII par la valeur 32 (0x20 en hexadécimal). La pseudo-instruction **lbu** (**load byte unsigned**) charge un byte depuis la mémoire dans les huit bits de poids faible d'un registre, et laisse les 24 bits de poids fort à zéro.

1. Décrivez en une phrase la fonction du programme. Que représente la valeur du registre **$v0** à la fin du programme (étiquette **finish**)?
   Donnez la valeur finale de **$v0** pour le cas suivant de contenu de mémoire (**$a0** = 1000):

   1000: 0x49 = "I"
   1001: 0x20 = " "
   1002: 0x20 = " "
   1003: 0x61 = "a"
   1004: 0x6d = "m"
   1005: 0x20 = " "
   1006: 0x00 = null

2. Que représente la valeur du registre **$t1** pendant l'exécution du programme? Pourquoi?

3. Quelle serait la conséquence si on changeait l'instruction **lbu** par **lb** dans le programme? (Rappelez-vous que **lb** charge un byte dans un registre en effectuant une extension de signe de 8 à 32 bits).

4. Le programme accède à un caractère à la fois avec l'instruction **lbu**. Modifiez le programme pour utiliser **lw** à la place de **lbu** et donc charger jusqu'à quatre caractères à la fois. Faites en sorte de minimiser autant que possible le nombre d'accès à la mémoire (c'est-à-dire, **lw** doit être exécuté à peu près 4 fois moins souvent que **lbu** dans le programme original). Considérez que le processeur est de type *little-endian*.

1. *Ce programme compte le nombre de mots que contient une chaîne de caractères et retourne le résultat dans $v0.*
   Pour l'exemple donné, le programme retourne $v0=2.

2. *$t1 est mis à '1' au début du programme et dès qu'un caractère "espace" est rencontré.*
   $t1 est mis à '0' dès que la première lettre d'un mot est rencontrée.

Init:	$t1 = '1'
"I"	$t1 = '0'
" "	$t1 = '1'
" "	$t1 = '1'
"a"	$t1 = '0'
"m"	$t1 = '0'
" "	$t1 = '1'

3. *Si l'on changeait l'instruction **lbu** par **lb**, le fonctionnement du programme serait totalement identique, car ce programme compare la valeur chargée uniquement avec 0 et 32 (espace).*

4. *Programme amélioré:*

```
prog:       add    $v0, $zero, $zero
            addi   $t0, $zero, 32
            addi   $t1, $zero, 1

loop:       lw     $t2, 0($a0)
            addi   $t4, $zero, 4
inner:      beq    $t4, $zero, nextword
            addi   $t4, $t4, -1
            andi   $t3, $t2, 0xFF

            beq    $t2, $zero, finish
            beq    $t2, $t0, space
            beq    $t1, $zero, next
            addi   $v0, $v0, 1
            add    $t1, $zero, $zero
            j      next

space:      addi   $t1, $zero, 1
next:       srl    $t2, $t2, 8
            j      inner

nextword:   addi   $a0, $a0, 4
            j      loop
finish:
```

Ecrivez, en assembleur MIPS, une fonction qui compare deux vecteurs de nombres signés de 32 bits. La fonction produit la valeur 0 si au moins un couple d'éléments partageant le même index dans les deux vecteurs diffère en valeur absolue de plus de 1000 (décimal). Dans le cas contraire, la fonction produit la valeur 1.

Au moment de l'appel de la fonction, le registre **$a0** contient l'adresse en mémoire du premier vecteur, le registre **$a1** contient l'adresse en mémoire du deuxième vecteur, et le registre **$a2** contient le nombre d'éléments dans chacun vecteur.

Remarques:

1. pour trouver la valeur absolue de la différence de deux nombres, il faut soustraire le plus petit du plus grand;
2. par simplicité, ignorez tout dépassement de capacité mais discutez brièvement ce qu'il faudrait changer dans le programme pour le prendre en compte.

1.

```
prog:  addi  $t6, $zero, 1000
       addi  $v0, $zero, 1

loop:  beq   $a2, $zero, finish
       lw    $t2, 0($a0)
       lw    $t3, 0($a1)
       slt   $t4, $t3, $t2          # if(t3<t2) t4=1
       bne   $t4, $zero, next       # if(t4=1) goto next
       sub   $t5, $t3, $t2
       j     next2

next:  sub   $t5, $t2, $t3

next2: slt   $t7, $t6, $t5          # if(t5>1000) t7=1
       bne   $t7, $zero, err        # if(t7!=0) goto err

       addi  $a0, $a0, 4
       addi  $a1, $a1, 4
       addi  $a2, $a2, -1
       j     loop

err:   add   $v0, $zero, $zero
finish:
```

2.

Il peut y avoir un dépassement de capacité lors de la soustraction, dans le cas où l'on soustrait une valeur négative à une valeur positive et que la différence est plus grande que la valeur max positive.

Exemple: 00'00'00'00 – 80'00'00'00; 7F'FF'FF'FF – FF'FF'FF'FF;

Pour éviter l'exception qui en découle, ***sub*** *peut être remplacé par* ***subu****.*

Dans ce cas, l'instruction ***slt $t7, $t6, $t5*** *ne couvre pas tous les cas et une manière de remédier à ce problème serait de remplacer l'instruction* ***slt*** *par* ***sltu****.*

Considérez le programme MIPS suivant (utilisez la figure 4.43 à la page 274 du COD si vous ne vous rappelez pas de la syntaxe MIPS):

```
      add     $t0, $a0, $zero
      add     $t1, $a1, $zero
      add     $t2, $a2, $a2
      add     $t2, $t2, $t2
      add     $t3, $t0, $t2
loop: lw      $t4, 0($t0)
      sw      $t4, 0($t1)
      addi    $t0, $t0, 4
      addi    $t1, $t1, 4
      sltu    $t5, $t0, $t3
      bne     $t5, $zero, loop
```

Assumez que, au début, les registres $a0 et $a1 stockent des adresses et le registre $a2 un nombre entier N. Les registres $t0 a $t5 sont utilisés pour stocker des valeurs temporaires et $zero est un registre qui vaut toujours zéro. Ajoutez des brefs commentaires à chaque ligne.

❶ Décrivez en une phrase la fonction du programme

❷ Pourquoi les instructions addi ajoutent-elles 4 à $t0 et $t1?

❸ Pourquoi l'instruction sltu (set less than unsigned) est-elle utilisée et non pas l'instruction slt?

Programme commenté:

```
     add   $t0, $a0, $zero    # $t0 <- a0
     add   $t1, $a1, $zero    # $t1 <- a1
     add   $t2, $a2, $a2      # $t2 <- 2*a2
     add   $t2, $t2, $t2      # $t2 <- 4*a2
     add   $t3, $t0, $t2      # $t3 <- a0 + *a2
loop:lw    $t4, 0($t0)        #
     sw    $t4, 0($t1)        #
     addi  $t0, $t0, 4        # $t0 <- $t0+4
                              # (pointe sur mot suivant)
     addi  $t1, $t1, 4        # $t1 <- $t1+4
                              # (pointe sur mot suivant)
     sltu  $t5, $t0, $t3      # if ($t0 < $t3)
                              # $t5 <- 1 else $t5 <- 0
     bne   $t5, $zero, loop   # if ($t5 != 0) go to loop
```

Le programme assembleur MIPS précédant copie la zone mémoire de l'adresse $a0 à $a0+4N dans la zone mémoire de l'adresse $a1 à $a1+4N. $t0 et $t1 sont les pointeurs des zones mémoire source et destination respectivement.

La mémoire d'un système MIPS est organisée par Byte et les mots (32-bits) sont alignés à partir d'une adresse qui est un multiple de 4. Les instructions LW et SW accèdent à des mots de 32-bits dont l'adresse est l'adresse la plus basse des 4-bytes.

Dans la ligne « sltu $t5, $t0, $t3 », on compare les registres $t0 et $t3 qui ont la fonction de pointeurs dans la zone mémoire source. La zone mémoire adressable du système se situe de l'adresse 00000000h à l'adresse FFFFFFFFh et un pointeur est donc toujours de type non signé.

Etudiez le programme MIPS suivant:

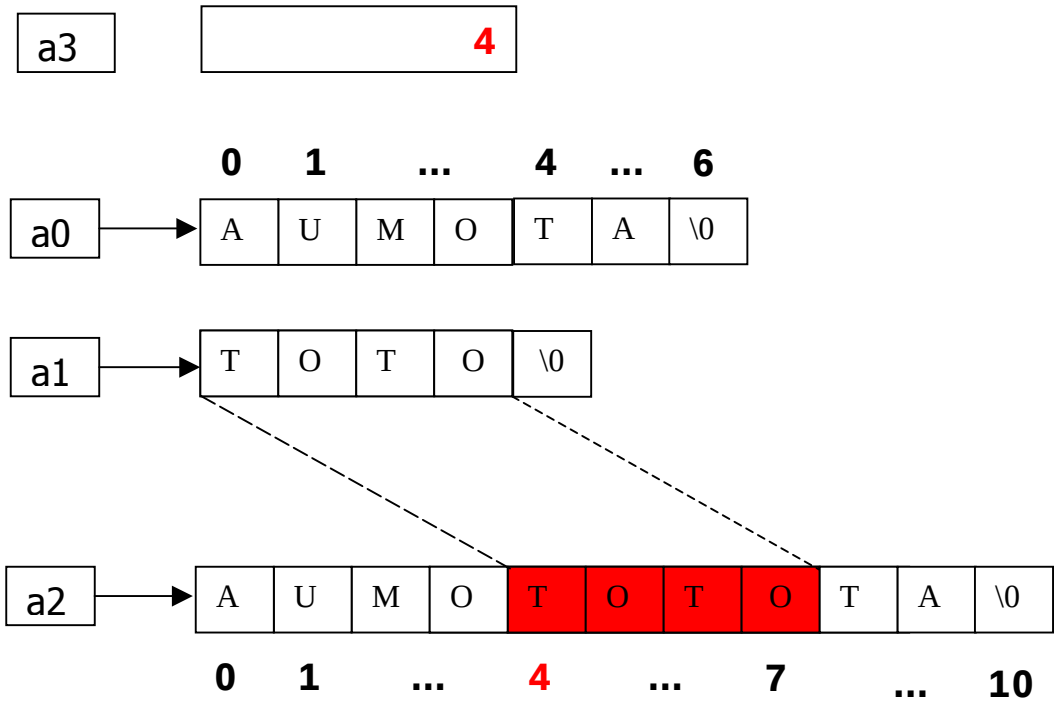
```
begin:    add  $t0, $a0, $zero
          add  $t1, $a1, $zero
          add  $t2, $a2, $zero
          add  $t3, $zero, $zero
          add  $t4, $zero, $zero
outer:    lbu  $t5, 0($t0)
          bne  $t3, $a3, cont
inner:    lbu  $t6, 0($t1)
          sb   $t6, 0($t2)
          addi $t1, $t1, 1
          addi $t2, $t2, 1
          addi $t4, $t4, 1
          bne  $t6, $zero, inner
          addi $t2, $t2, -1
          addi $t4, $t4, -1
cont:     sb   $t5, 0($t2)
          addi $t0, $t0, 1
          addi $t2, $t2, 1
          addi $t3, $t3, 1
          bne  $t5, $zero, outer
          addi $t3, $t3, -1
          add  $v0, $t3, $t4
fin:      jr   $ra
```

a. Décrivez en une phrase la fonction de ce programme, en sachant qu'il prend quatre arguments dans les registres $a0, $a1, $a2 et $a3. Les arguments $a0 et $a1 correspondent aux adresses en mémoire de deux chaînes de caractères terminées par des caractères NULL — c'est-à-dire, des octets *(bytes)* à 0.

b. Modifiez le programme de façon à ne plus utiliser l'instruction lbu mais seulement l'instruction lw tout en gardant exactement la même fonctionnalité. Considérez que les adresses des débuts des chaînes sont « alignées », c'est-à-dire qu'elles sont toujours des multiples de 4, et que le processeur est *little-endian*. L'instruction sb reste disponible.

c. Est-ce que le programme du point b. devrait être adapté si le processeur était *big-endian* ? Si oui, décrivez brièvement les modifications qu'il faudrait apporter.

a)
The program returns a string pointed by $a2 that contains the string pointed by $a0 with the string pointed by $a1 inserted at the position indexed by $a3.



b)
The following subroutine does not use the lbu instruction. Instead, it uses the lw instruction having in mind that the processor is *little-endian*.

```
begin:    add  $t0, $zero, $a0
          add  $t1, $zero, $a1
          add  $t2, $zero, $a2
          add  $t3, $zero, $zero
          add  $t4, $zero, $zero
outer:    andi $t7, $t3, 0x3
          bne  $t7, $zero, no_ld1
          lw   $t5, 0($t0)
          addi $t0, $t0, 4
no_ld1:   andi $t7, $t5, 0xff      # ***
          srl  $t5, $t5, 8         # ***
          bne  $t3, $a3, cont
inner:    andi $t8, $t4, 0x3
          bne  $t8, $zero, no_ld2
          lw   $t6, 0($t1)
```

```
          addi $t1, $t1, 4
no_ld2:   andi $t8, $t6, 0xff      # +++
          srl  $t6, $t6, 8         # +++
          sb   $t8, 0($t2)
          addi $t2, $t2, 1
          addi $t4, $t4, 1
          bne  $t8, $zero, inner
          addi $t2, $t2, -1
          addi $t4, $t4, -1
cont:     sb   $t7, 0($t2)
          addi $t2, $t2, 1
          addi $t3, $t3, 1
          bne  $t7, $zero, outer
          addi $t3, $t3, -1
          add  $v0, $t3, $t4

fin:      jr   $ra
```

c)
Yes. Instead of the sequence of andi and srl (marked by *** and +++) the following instruction sequence might be used (alternate solutions are possible):

```
          srl  $t7, $t5, 24   # prepare byte
          andi $t7, $t7, 0xff # get byte
          sll  $t5, $t5, 8    # prepare next

          ...

          srl  $t8, $t6, 24   # prepare byte
          andi $t8, $t8, 0xff # get byte
          sll  $t6, $t6, 8    # prepare for next
```

Ecrivez une fonction MIPS qui produit un vecteur C de nombres 32-bit signés à partir de deux vecteurs A et B, aussi composés de nombres 32-bits signés et en complément à deux. Le vecteur C contient la valeur absolue de la différence de A et B :

$$\vec{C} = \left| \vec{A} - \vec{B} \right|$$

Les vecteurs A, B et C sont pointés respectivement par les registres $a0, $a1 et $a2. Le registre $a3 indique le nombre d'éléments contenus dans chaque vecteur.

a. Ecrire la fonction en ignorant toute possibilité de dépassement de capacité. Aussi, considérez une version de MIPS dont les instructions de soustraction ou de négation arithmétique ne sont pas disponibles. Toutes les opérations logiques sont, par contre, disponibles (and, or, xor, not, etc.).

b. Discutez les possibilités de dépassement de capacité. Quelles instructions de votre programme pourraient générer des dépassements de capacité ? Décrivez brièvement (sans nécessairement modifier le programme) comment détecter ces dépassements.

a) Since neither arithmetic negation nor subtraction is available, bitwise negation and addition must be used in order to implement the requested function.

```
begin:    add  $t0, $zero, $a0     # t0 <- a0
          add  $t1, $zero, $a1     # t1 <- a1
          add  $t2, $zero, $a2     # t2 <- a2
          add  $t3, $zero, $a3     # t3 <- a3
loop:     beq  $t3, $zero, fin     # if t3 = 0 then
goto
                                   # finish
          lw   $t4, 0($t0)         # t4 <- mem[t0]
          lw   $t5, 0($t1)         # t5 <- mem[t1]
          not  $t5, $t5            # t5 <- not t5
          addi $t5, $t5, 1         # t5 <- t5 + 1 (***)
          add  $t5, $t4, $t5       # t5 <- t4 + t5 (+++)
          slt  $t4, $t5, $zero     # check the sign
          bne  $t4, $zero, skip  #if positive goto skip
          not  $t5, $t5              # t5 <- t5
          addi $t5, $t5,             # t5 <- t5 + 1 (***)
skip:     sw   $t5, 0($t2)         # mem[t2] <- t5
          addi $t0, $t0, 4         # t0 <- t0 + 4
          addi $t1, $t1, 4         # t1 <- t1 + 4
          addi $t2, $t2, 4         # t2 <- t2 + 4
          addi $t3, $t3, -1        # t3 <- t3 - 1
          j    loop

fin:      jr   $ra
```

b) The instructions marked by *** and +++ can cause an overflow.

In the *** case, the overflow will appear if the register $t5 contains the smallest negative number (1 at the MSB followed by 0s) before the appropriate NOT instruction is performed. To detect the smallest negative number, find out that the number is negative and then reset the sign bit and check if there are only 0s remaining.

In the +++ case, there is standard overflow treatment (see COD.

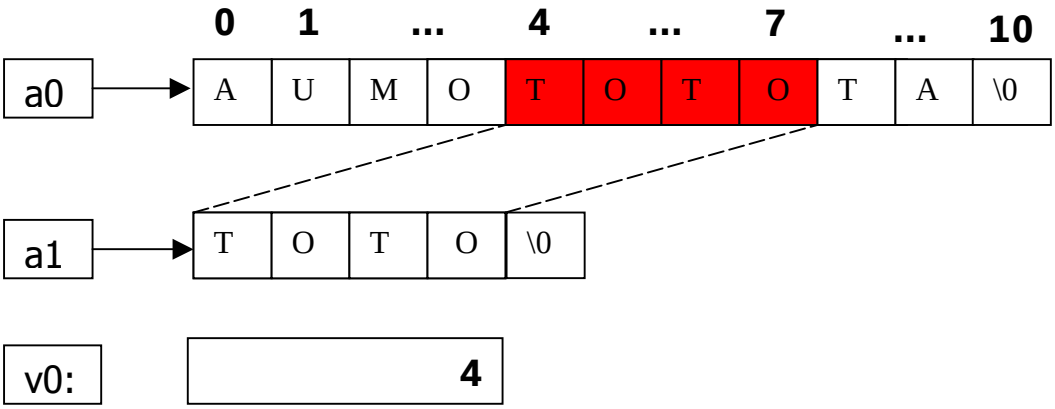
Etudiez le programme MIPS suivant :

```
begin :   move $t0, $a0
          move $t1, $a1
          move $t2, $zero
          addi $v0, $zero, -1
cont :    lbu  $t4, 0($t1)
outer :   lbu  $t3, 0($t0)
          beq  $t3, $zero, fin
          bne  $t3, $t4, skip
          move $v0, $t2
          move $t5, $t0
inner :   addi $t0, $t0, 1
          addi $t1, $t1, 1
          lbu  $t3, 0($t0)
          lbu  $t4, 0($t1)
          beq  $t4, $zero, fin
          beq  $t3, $zero, fail
          beq  $t3, $t4, inner
          addi $t0, $t5, 1
          move $t1, $a1
          addi $v0, $zero, -1
          j    cont

skip :    addi $t0, $t0, 1
          addi $t2, $t2, 1
          j    outer

fail :    addi $v0, $zero, -1
fin :     jr   $ra
```

a. Décrivez en une phrase la fonction de cette fonction, sachant qu'elle prend deux arguments dans les registres $a0 et $a1. Ces deux arguments correspondent aux adresses en mémoire de deux chaînes de caractères terminées par des caractères NULL — c'est-à-dire, des octets *(bytes)* à 0.
b. Expliquez en quelques mots ce que le contenu du registre $t5 représente et la situation dans laquelle cette valeur est nécessaire.
c. Supposez ne pas avoir d'instruction lbu disponible, mais seulement l'instruction lw pour lire la mémoire. Ecrivez une fonction qui implémente la même fonctionnalité que

l'instruction lbu $v0, 0($a0) : à l'appel $a0 contient l'adresse d'un octet dans la mémoire et à la fin de l'appel $v0 doit contenir l'octet correspondant (considérez un processeur *big-endian*). Rappelez-vous que l'instruction lbu ne fait pas d'extension de signe et faites en sorte que les adresses passées à lw soient toujours « alignées », c'est-à-dire qu'elles soient toujours des multiples de 4.

a)
The subroutine returns the position (index) of the first appearance of a substring pointed by $a1 in the input string pointed by $a0. If the substring is not found, the negative position is returned (-1).



```
begin:    move $t0, $a0          # t0 <- a0
          move $t1, $a1          # t1 <- a1
          move $t2, $zero        # t2 <- 0
          addi $v0, $zero, -1    # v0 <- -1
cont:     lbu  $t4, 0($t1)       # t4 <- mem[t1]
outer:    lbu  $t3, 0($t0)       # t3 <- mem[t0]
          beq  $t3, $zero, fin   # if t3 = 0 goto fin
          bne  $t3, $t4, skip    # if t3 <> t4 goto
                                 # skip
          move $v0, $t2          # remember index
          move $t5, $t0          # wrong guess backup
inner:    addi $t0, $t0, 1       # t0 <- t0 + 1
          addi $t1, $t1, 1       # t1 <- t1 + 1
          lbu  $t3, 0($t0)       # t3 <- mem[t0]
          lbu  $t4, 0($t1)       # t4 <- mem[t1]
          beq  $t4, $zero, fin   # return  found
          beq  $t3, $zero, fail  # if t3 = 0 fail
          beq  $t3, $t4, inner   # continue inner
                                 # loop
          addi $t0, $t5, 1       # recover wrong
                                 # guess
          move $t1, $a1          # recover

          j    cont              # goto continue
skip:     addi $t0, $t0, 1       # t0 <- t0 + 1
          addi $t2, $t2, 1       # t2 <- t2 + 1
          j    outer             # goto outer

fail:     addi $v0, $zero, -1    # v0 <- -1
fin:      jr   $ra
```

b)
Register $t5 enables recovery from a "wrong guess". It holds the address where the matching started. If it does not succeed (the substring is not found), the matching is restarted from the next potential match at the address $t5 + 1.

c)
The following subroutine provides the functionality of the lbu instruction. Two possible solutions are presented. Both of them assume a *big-endian* processor.

```
xlbu:     lui  $t1, 0xffff          # t1 <- 0xffff0000
          addi $t1, $t1, 0xfffc     # t1 <- 0xfffffffc
          and  $t0, $a0, $t1        # t1 <- a0 & t1
                                    # (align)
          lui  $t1, 0xff00          # t1 <- 0xff000000
          and  $t2, $a0, 0x3        # t2 <- a0 & 0x3
                                    # (get offset)
          lw   $t3, 0($t0)          # t3 <- mem[$t0]
loop:     beq  $t2, $zero, done     # if t2 = 0 done
          sll  $t3, $t3, 8          # t3 <- t3 << 8
                                    # (next byte)
          addi $t2, $t2, -1         # t2 <- t2 - 1
          j    loop                 # goto loop

done:     and  $t3, $t3, $t1        # t3 <- t3 &
                                    # 0xff000000
          srl  $v0, $t3, 24         # v0 <- t3 >> 24
          jr   $ra                  # return
```

Another possible solution that uses srl:

```
xlbu:     lui  $t1, 0xffff          # t1 <- 0xffff0000
          addi $t1, $t1, 0xfffc     # t1 <- 0xfffffffc
          and  $t0, $a0, $t1        # t1 <- a0 & t1
                                    # (align)
          and  $t2, $a0, 0x3        # t2 <- a0 & 0x3
                                    # (get offset)
          lw   $t3, 0($t0)          # t3 <- mem[t0]
loop:     sltiu $t1, $t2, 3         # check t2 < 3
          beq  $t1, $zero, done     # if t2 = 3 done
          srl  $t3, $t3, 8          # t3 <- t3 >> 8
                                    # (next byte)
          addi $t2, $t2, 1          # t2 <- t2 + 1
          j    loop                 # goto loop

done:     and  v0, t3, 0xff         # v0 <- t3 & 0xff
                                    # (get byte)
          jr   $ra                  # return
```

Ecrivez une fonction MIPS qui effectue une conversion de la représentation BCD *(Binary Coded Decimal)* à la représentation binaire ordinaire. Dans la représentation BCD, les nombres sont codés à partir de leur représentation décimale ; chaque chiffre décimal occupe 4 bits et est code en binaire. Par exemple, la valeur 1992 en décimal est codée en BCD sur 16 bit comme 0001 1001 1001 0010, alors que sa représentation binaire ordinaire est 0000 0111 1100 1000. Remarquez que certaines valeurs binaires comme 1111 1010 1100 1110 ne peuvent pas représenter des valeurs BCD ; cela arrive dès qu'un groupe de 4 bits représente une valeur binaire plus grands que 9. La valeur 32 bits non signée à convertir se trouve dans le registre $a0 et le résultat binaire à la fin de la fonction doit se trouver dans le registre $v0. Si le nombre contenu en $a0 ne peut pas représenter un nombre en format BCD, le registre $v0 doit contenir -1 à la fin de l'exécution.

a. Ecrire la fonction de conversion en ignorant toute possibilité de dépassement de capacité et en considérant disponibles des instructions de multiplication sur 32 bits :

```
xmul      rd, rs, rt
xmuli     rd, rs, imm
```

b. Les instructions xmul et xmuli n'existent pas dans MIPS. Modifiez le programme de façon à ne plus utiliser d'instruction de multiplication.

c. Discutez les possibilités de dépassement de capacité. Modifiez le programme, si nécessaire.

a)
In order to simplify the computation, the conversion is done using the following decomposition (as an example, a four digit BCD number is used – *abcd*):

$abcd_{10} = a*10^3 + b*10^2 + c*10^1 + d*10^0 =$
$= (((a*10 + b)*10 + c)*10 + d)$

An eight digit BCD number is converted using the following procedure:

```
begin:    add  $t0, $a0, $zero     # init t0
          add  $v0, $v0, $zero     # init v0
          beq  $t0, $zero, fin     # finish if zero
                                   # (optional)
          addi $t1, $zero, 8       # number of digits
loop:     srl  $t2, $t0, 28        # get the MS digit d
          sltiu $t3, $t2, 10       # check d < 10
          bne  $t3, $zero, skip    # skip if ok
          addi $v0, $zero, -1      # if error,
          j    fin                 # finish
skip:     xmuli $v0, $v0, 10       # v0 <- v0*10
          add  $v0, $v0, $t2       # v0 <- v0 + d
          sll  $t0, $t0, 4         # prepare next digit
          addi $t1, $t1, -1        # decrement counter
          bne  $t1, $zero, loop    # continue until 0

fin:      jr   $ra                 # return
```

b)
Multiplication by 10 can be replaced by shifting and adding (the idea is X * 10 = X*8 + X*2). The following is the MIPS implementation:

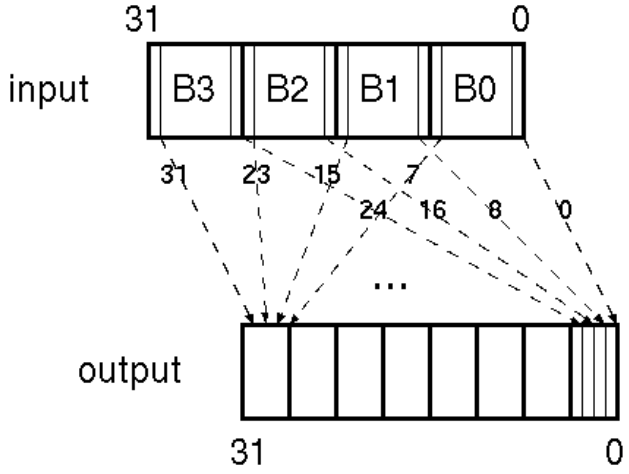
```
          ...

skip:     sll  $t3, $v0, 3          # mul v0 by 8
          sll  $t4, $v0, 1          # mul v0 by 2
          add  $v0, $t3, $t4        # v0 <- 10 * v0

          ...
```

c)
It is not possible to have an overflow since the largest number in 8 digit BCD representation (32 bit) is smaller than the largest 32-bit unsigned number (or the largest number in 2's complement representation).

On utilise un processeur MIPS pour contrôler la transmission de caractères ASCII (8 bits par caractère: 7 bits d'information + 1 bit de parité) sur une ligne série. La ligne est particulièrement exposée aux erreurs de type « burst » (un burst d'erreurs sur la ligne de transmission affecte plusieurs bits consécutifs). Pour améliorer la détection des erreurs, on utilise la technique d' « interleaving » avant la transmission :

1. On groupe 4 caractères à transmetttre dans un bloc de 32 bits (4x8 bits). On peut donc assigner une position absolue (i.e., 0..31) à chaque bit dans le bloc de 32 bits. Par exemple, le bit 5 du byte 2 aura la position 21 (5 + 2 x 8), le bit 0 du byte 3 la position 24 (0 + 3 x 8).

2. On change l'ordre des bits et on forme un nouveau bloc de 32 bits pour la transmission de la façon suivante :



D'après la figure, on peut définir la fonction **interleaving** qui détermine la position d'un bit après interleaving (i = 0..31 représente la position originale) :

$$newpos(i) = i \% 8 + (i \ mod \ 8) * 4$$

L'opération % représente la division entre nombres entiers et *mod* représente le modulo ou reste de la division entière. Par exemple, après interleaving le bit 23 (position originale) aura la nouvelle

position 30 (23 % 8 + (23 *mod* 8) x 4 = 2 + 28) et le bit 24 la nouvelle position 3 (24 % 8 + (24 *mod* 8) x 4 = 3 + 0).

a. Ecrivez une fonction MIPS qui implémente la fonction newpos(i). Le paramètre i est passé à la fonction dans le registre **$a0** et la fonction rend la valeur calculée dans le registre **$v0**.

b. En utilisant la fonction **newpos**, écrivez un programme MIPS pour réaliser l'interleaving d'un bloc de 4 bytes (i.e., 32 bits, comme dans la figure ci-dessus). Le bloc source se trouve dans le registre **$a0**. Le résultat du programme doit être placé dans le registre **$v0**. Les instructions **sllv** (*shift left logical variable*) et **srlv** (*shift right logical variable*) décalent le contenu du registre **rs** de la quantité spécifiée dans le registre **rd** et placent le résultat dans le registre **rt** :

```
sllv   rt, rs, rd
srlv   rt, rs, rd
```

c. Discutez quelles sont les conventions MIPS par rapport à la sauvegarde des registres entre appels de fonction. Discutez leur application aux fonctions écrites aux points précédents.

d. Supposez que l'on envoie de l'information entre un ordinateur source *little endian* et un ordinateur destination *big endian*. Faut-il faire des modifications au programme qui effectue l'interleaving ? Si oui, discutez brièvement les modifications nécessaires. Si non, expliquez la situation.

a)

```
newpos:   add  $t1, $a0, $zero     ; t1 <- a0
          srl  $t2, $t1, 3         ; t2 <- t1 % 8
          andi $t1, $t1, 0x7       ; t1 <- t1 mod 8
          sll  $t1, $t1, 2         ; t1 <- t1 * 4
          add  $v0, $t1, $t2       ; v0 <- t1 + t2
          jr   $ra
```

b)

```
prog:     add  $t0, $a0, $zero     ; t0 <- a0
          add  $t1, $zero, $zero   ; t1 <- 0, result
          add  $t2, $zero, $zero   ; t2 <- 0, index i
          addi $t3, $zero, 32      ; t3 <- 32

loop:     andi $t4, $t0, 0x1       ; test bit0
          beq  $t4, $zero, skip    ; skip if bit0 = 0
          add  $a0, $t2, $zero     ; a0 <- i
          jal  newpos              ; call newpos(i)
          addi $t5, $zero, 0x1     ; t5 <- 1
          sllv $t5, $t5, $v0       ; shift to newpos(i)
          or   $t1, $t1, $t5       ; set bit at newpos(i)
skip:     srl  $t0, $t0, 1         ; shift to next bit
          addi $t2, $t2, 1         ; t2 <- t2 + 1
          bne  $t2, $t3, loop      ; if t2 != t3 goto loop

fin :     add  $v0, $t1, $zero     ; v0 <- t1
```

c) L'une des conventions est que la routine appelée sauvegarde les registres (le registre d'adresse de retour, plus les registres utilisés par la routine, y exclus les registres temporaires) sur la pile, tout au début de son exécution. La routine restaure les registres depuis la pile, en fin de son exécution.

Appliquez sur les routines dans a) et b), on voit que, même si le fonctionnement est correct, on perd les contenus originaux de $ra et $a0 dans le programme principal (après l'appel au newpos). Ce n'est pas un problème dans le cas d'un programme *stand-alone*, mais, dans le cas où le programme ci-dessus est appelé d'ailleurs, ça empêche l'exécution normale. Avec une pile qui s'augmente vers les adresses descendantes et le pointeur qui pointe sur le sommet de la pile, on aurait le code de sauvegarde et rétablissement suivant :

```
prog:     sw   $ra, -4($sp)   ; mem[sp-4] <- ra
          sw   $a0, -8($sp)   ; mem[sp-8] <- a0
          addi $sp, $sp, -8   ; sp <- sp – 8
          ...

          ...
fin:      lw   $a0, 0($sp)    ; a0 <- mem[sp]
          lw   $ra, 4($sp)    ; ra <- mem[sp+4]
          addi $sp, $sp, 8    ; sp <- sp + 8
```

d) Le fonctionnement correct du programme dépend de « *endianess* ». Si un ordinateur destination *big endian* reçoit les paquets de 4 bytes d'un ordinateur *little endian* (l'ordre avant « interleaving » $B_3B_2B_1B_0$), il doit les interpréter correctement. C'est-à-dire, il doit d'abord faire « *deinterleaving* » (1) et, en suite, établir l'ordre correct $B_3B_2B_1B_0$ (2) avant stocker les données dans la mémoire. En ayant une fonction de « *deinterleaving* » élégante disponible, on peut éviter le rétablissement de l'ordre (2). Ecrire la fonction.

Considérez le programme MIPS suivant:

```
start:      addi    $t1, $a1, -1
outer:      beq     $t1, $zero, fin
            add     $t0, $a0, $zero
            add     $t2, $t1, $zero
inner:      beq     $t2, $zero, cont
            lw      $t3, 0($t0)
            lw      $t4, 4($t0)
            sltu    $t5, $t4, $t3
            beq     $t5, $zero, skip
            sw      $t3, 4($t0)
            sw      $t4, 0($t0)
skip:       addi    $t0, $t0, 4
            addi    $t2, $t2, -1
            j       inner

cont:       addi    $t1, $t1, -1
            j       outer
fin:
```

a. Décrivez en une phrase la fonction du programme, sachant que les deux arguments chargés dans **$a0** et **$a1** contiennent respectivement l'adresse d'un tableau dans la mémoire et le nombre de ses éléments.

b. Expliquez en quelques mots quelle est la fonction des deux instructions **lw** et des deux instructions **sw** dans ce contexte.

c. Quel est le type des valeurs traitées? Justifier votre réponse.

d. Expliquez en quelques mots ce que le contenu de trois registres **$t0**, **$t1** et **$t2** représente. Corrigez le programme pour traiter correctement le cas **$a1** = 0.

e. Alors que le programme original ne rend aucun résultat, on veut que la fonction rende la valeur suivante dans le registre **$v0** :

   - 0 dans le cas où aucune donnée dans la mémoire n'a été changée ;
   - –1 dans le cas où au moins une donnée dans la mémoire a été changée.

   Modifier le programme pour qu'il soit possible de l'appeler comme une fonction standard MIPS.

a) Ce programme trie les éléments d'un vecteur dans l'ordre ascendant.

b) Les deux instructions lw et les deux instructions sw sont utilisées pour échanger la position (« *swap* ») de deux éléments dans la mémoire.

c) Le programme traite les nombres entiers, non signés, comme la comparaison est faite par l'instruction sltu (« *set-less-than-unsigned* »).

d) Le registre $t0 est utilisé comme le pointeur sur les éléments du vecteur. Les registres $t1 et $t2 sont utilisés comme les compteurs, pour gérer les deux boucles. Dans chaque moment, le registre $t1 contient le nombre de fois à répéter la boucle « *inner* » (autrement dit, le nombre d'itérations de la boucle « *outer* »). Le registre $t2 détermine le nombre d'itérations de la boucle « *inner* ».

Dans le cas $a1 = 0, le programme ne fonctionnera pas correctement (on risque d'avoir le programme qui génère les adresses fausses). Il faut donc éviter ce cas :

```
start: beq   $a1, $zero, fin
       addi  $t1, $a1, -1
       ...
```

e) On suppose que la pile croît vers les adresses descendantes et que le registre $sp pointe sur le mot suivant la première place libre sur la pile.

```
start:          sw      $ra, 0($sp)     ; mem[sp] <- ra
                addi    $sp, $sp, -4    ; sp <- sp - 4
                add     $v0, $zero, $   ; v0 <- 0
                addi    $t1, $a1, -1
outer:          beq     $t1, $zero, fin

                ...                     ; no changes

                sw      $t3, 4($t0)
                sw      $t4, 0($t0)
                addi    $v0, $zero, -1 ; v0 <- -1
skip:           addi    $t0, $t0, 4
                addi    $t2, $t2, -1
                j       inner

cont:           addi    $t1, $t1, -1
                j       outer
fin:            lw      $ra, 4($sp)     ; ra <- mem[sp+4]
                addi    $sp, $sp, 4     ; sp <- sp + 4
                jr      $ra
```

Analysez la fonction MIPS suivante:

```
func:       sll   $t0, $a1, 2
            add   $t0, $a0, $t0
            addi  $t0, $t0, -4
loop:       slt   $t1, $a0, $t0
            beq   $t1, $zero, fin
            lw    $t2, 0($a0)
            lw    $t3, 0($t0)
            sw    $t2, 0($t0)
            sw    $t3, 0($a0)
            addi  $t0, $t0, -4
            addi  $a0, $a0, 4
            j     loop

fin:       jr    $ra
```

Lors de l'appel, **$a0** contient l'adresse en mémoire d'un vecteur de nombres codés sur 32 bits et **$a1** contient un nombre entier.

a. Décrivez en une phrase la fonction du programme.

b. Les nombres qui composent le vecteur doivent-ils forcement être signés ou non signés ? Ou bien les deux cas de figure sont-ils possibles? Expliquez brièvement mais précisément votre réponse.

c. On désire adapter ce programme pour manipuler des octets. Pour ceci, on a besoin d'une fonction qui reçoit en **$a0** quatre octets et qui rend les mêmes quatre octets en **$v0** dans l'ordre inverse : l'octet aux bits 31-24 passe aux bits 7-0, celui à 23-16 passe à 15-8, etc. Ecrivez une telle fonction en respectant les conventions MIPS ordinaires.

a)
La fonction de ce programme est d'inverser l'ordre des éléments d'un vecteur sur la memoire. L'analyse de cette fonction est la suivante :

- **a0 :** pointe à l'élément suivant du vecteur *depuis le début* qui doit être échangé
- **t0 :** pointe à l'élément suivant du vecteur *depuis la fin* qui doit être échangé

```
func:  sll   $t0, $a1, 2      ; $t0 = Nombred'élém.*sizeof(élém.)
       add   $t0, $a0, $t0    ; $t0 pointe à la fin du vecteur
       addi  $t0, $t0, -4     ; $t0 pointe au dernier élément
loop:  slt   $t1, $a0, $t0
       beq   $t1, $zero, fin ; if( $a0 >= $t0) go to fin
       lw    $t2, 0($a0)     ; Valeur pointé par a0 dans t2
       lw    $t3, 0($t0)     ; Valeur pointé par t0 dans t3
       sw    $t2, 0($t0)     ; Copie t2 à l'address pointé par t0
       sw    $t3, 0($a0)     ; Copie t3 à l'address pointé par a0
       addi  $t0, $t0, -4    ; Met à jour le pointeur de fin
       addi  $a0, $a0, 4     ; Met à jour le pointeur de début
       j     loop            ; Continue la boucle

fin:   jr    $ra             ; Retour de la fonction
```

b)
La fonction MIPS ne modifie pas d'éléments du vecteur, donc il n'y a aucune restriction sur le contenu des éléments du vecteur. Les éléments peut être signés, non-signés ou des nombres d'autre interprétation.

c)
La fonction qui inverse l'ordre des octet d'une donnée sur 32 bit est la suivante:

```
Inv_byte: addi $t0, $zero, 0xFF; Masque d'octet dans t0
     .    and  $t1, $a0, $t0   ; Octet 7-0 dans t1
     .....sll  $t1, $t1, 24    ; Décale octet 7-0 à octet 31-24
     .....add  $v0, $zero, $t1 ; Octet 31-24 de résultat est prêt

     .....sll  $t0, $t0, 8     ; Masque d'octet 15-8 dans t0
....      and  $t1, $a0, $t0   ; Octet 15-8 dans t1
     .....sll  $t1, $t1, 8     ; Décale octet 15-8 à octet 23-16
     .....or   $v0, $v0, $t1   ; Octet 23-16 de résultat est prêt

     .....sll  $t0, $t0, 8     ; Masque d'octet 23-16 dans t0
....      and  $t1, $a0, $t0   ; Octet 23-16 dans t1
     .....srl  $t1, $t1, 8     ; Décale octet 23-16 à octet 15-8
     .....or   $v0, $v0, $t1   ; Octet 15-8 de résultat est prêt

     .....sll  $t0, $t0, 8     ; Masque d'octet 31-24 dans t0
....      and  $t1, $a0, $t0   ; Octet 31-24 dans t1
     .....srl  $t1, $t1, 24    ; Décale octet 31-24 à octet 7-0
     .....or   $v0, $v0, $t1   ; Octet 7-0 de résultat est prêt

fin:      jr   $ra             ; Retour de la fonction
```

Analysez la fonction MIPS suivante:

```
func:     add  $t0, $zero, $zero
          add  $t1, $zero, $a0
          lw   $t2, 0($a0)
          lw   $t3, 0($a0)
label:    lw   $t5, 0($t1)
          slt  $t4, $t2, $t5
          bne  $t4, $zero, cont1
          add  $t2, $zero, $t5
cont1:    slt  $t4, $t5, $t3
          bne  $t4, $zero, cont2
          add  $t3, $zero, $t5
cont2:    addi $t0, $t0, 1
          addi $t1, $t1, 4
          bne  $a1, $t0, label
          add  $t4, $t2, $t3
          sra  $v0, $t4, 1
fin:      jr   $ra
```

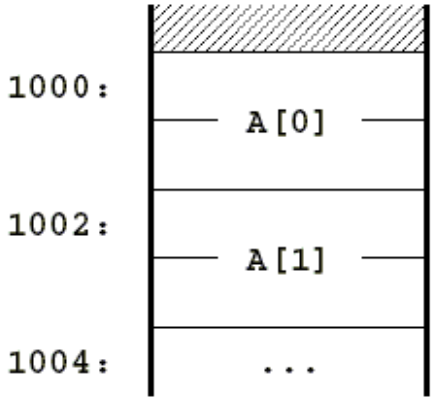
Lors de l'appel, **$a0** contient l'adresse en mémoire d'un vecteur de nombres codés sur 32 bits et **$a1** contient un nombre entier. **$v0** est la valeur de retour de la fonction.

a. Décrivez en une phrase la fonction du programme. Quelle est la valeur retournée par le programme ? Est-ce que cette valeur est toujours exacte?

b. Est-ce que les nombres composant le vecteur doivent être forcement signés ou non signés, ou les deux cas de figure sont possibles? Expliquez brièvement mais précisément votre réponse. Si seulement un type de nombres (signés ou non signés) peut être traité par la fonction telle qu'elle est écrite, indiquez toutes les modifications nécessaires pour l'adapter à l'autre type.

c. Adaptez la fonction pour traiter un vecteur de nombres entiers codés sur 16 bits au lieu de 32 bits. Les demi-mots de 16 bits

sont placés dans la mémoire selon l'exemple suivant. Supposez un processeur *big-endian* et minimisez le nombre d'accès mémoire (utilisez donc **`lw`** pour accéder à la mémoire). Supposez également que **`$a0`** est toujours un multiple de 4 à l'appel de la fonction.



a)

La fonction du programme est de trouver la moyenne entre le maximum et le minimum éléments du vecteur pointé par **$a0**. La valeur retournée est également la moyenne entre le maximum et le minimum éléments du vecteur pointé par **$a0**. Comme la division par 2 (la dernière commande avant le retour de la fonction) est une division des nombres entiers, la moyenne obtenue n'est pas toujours exacte. Une explication detaillée du programme est la suivante:

```
func:  add  $t0, $zero, $zero  ; Initialise compteur d'elem.(t0)
       add  $t1, $zero, $a0    ; t1 pointe au début du vecteur
       lw   $t2, 0($a0)        ; Initialise le min (t2)
       lw   $t3, 0($a0)        ; Initialise le max (t3)
label: lw   $t5, 0($t1)        ; Le prochain elem. dans t5
       slt  $t4, $t2, $t5      ; ($t2 >= $t5 ) => $t4 = 0
       bne  $t4, $zero, cont1  ;
       add  $t2, $zero, $t5    ; Met à jour le min dans t2
cont1: slt  $t4, $t5, $t3      ; ($t5 >= $t3 ) => $t4 = 0
       bne  $t4, $zero, cont2  ;
       add  $t3, $zero, $t5    ; Met à jour le max dans t3
cont2: addi $t0, $t0, 1        ;
       addi $t1, $t1, 4        ; Pass au prochain élément
       bne  $a1, $t0, label    ; Vérifie si la fin du vecteur
       add  $t4, $t2, $t3      ; Additionne le max et le min
       sra  $v0, $t4, 1        ; Division de la somme par 2
fin:   jr   $ra                ; Retour de la fonction
```

b)

Les composants du vecteur pointé par **$a0** doivent être nécessairement signés par le simple fait que les comparaisons sur des nombres signés est utilisés pour trouver le maximum est le minimum éléments du vecteur. Il y a aussi le calcul de la moyenne qui est effectué pour les nombres signés. Donc, les modifications à faire sont les suivantes:

```
...
slt  $t4, $t2, $t5      =>   sltu  $t4, $t2, $t5
...
slt  $t4, $t5, $t3   .. =>   sltu  $t4, $t5, $t3
...
add  $t4, $t2, $t3      =>   addu  $t4, $t2, $t3
sra  $v0, $t4, 1        =>   srl  $v0, $t4, 1
```

c)

Il est demandé de minimiser les accès à la memoire. Ainsi, lors de chaque lecture sur 32 bit (avec la commande **lw**) de la memoire, il faut sauvegarder le contenu dans un registre pour ne pas avoir besoin de le lire une seconde fois (puisqu'il y a deux élément du vecteur stocké dans ces 32 bits). Dans la solution suivante, le registre **t6** est utilisé dans ce but. Le premier élément du vecteur est utilisé pour initialiser le min et le max et ensuite le même élément est comparé avec le min et le max comme tous les autres éléments du vecteur. De plus, un fanion indique si l'élément est sur les 16 bit de poids fort ou sur les 16 bit de poids faible.

```
func: lw   $t2, 0($a0)       ; Premier 2 elem. du vect. dans t2
      add  $t6, $zero, $t2   ; Premier 2 elem. du vec. dans t6
      sra  $t2, $t2, 16      ; Initialise min avec premier elem.
      add  $t3, $zero, $t2   ; Initialise max avec premier elem.
      addi $t0, $zero, 0x1   ; Initialise compteur d'elem.(t0)
      add  $t1, $zero, $a0   ; t1 pointe au début du vecteur
      add  $t7, $zero, $zero ; Initialise le fanion
loop: beq  $t7, $zero, LSW   ; Fanion indique poids faible?
      lw   $t5, 0($t1)       ; Lecture de 2 elem. suivantes
      add  $t6, $zero, $t5   ; Une copie du contenu dans t6
      j    MSW               ; L'elem. suivant sur poids fort
LSW:  sll  $t5, $t6, 1       ; Decale sur le poids fort
MSW:  sra  $t5, $t5, 16      ; Decale sur poids faible avec signe
      slt  $t4, $t2, $t5     ; ($t2 >= $t5 ) => $t4 = 0
      bne  $t4, $zero, cont1;
      add  $t2, $zero, $t5   ; Met à jour le min dans t2
cont1:slt  $t4, $t5, $t3     ;($t5 >= $t3 ) => $t4 = 0
      bne  $t4, $zero, cont2;
      add  $t3, $zero, $t5   ; Met à jour le max dans t3
cont2:addi $t0, $t0, 0x1     ; Pass au prochain élément
      nor  $t7, $t7, $t7     ; Met à jour le fanion
      beq  $t7, $zero, cont3;
      addi $t1, $t1, 0x4     ; Met à jour pointeur du vect.
cont3:bne  $a1, $t0, loop    ; Vérifie si la fin du vecteur
      add  $t4, $t2, $t3     ; Additionne le max et le min
      sra  $v0, $t4, 0x1     ; Division de la somme par 2
fin:  jr   $ra               ;Retour de la fonction
```

L'algorithme CORDIC (« *COordinate Rotation DIgital Computer* ») permet de calculer la longueur d'un vecteur *V(X, Y)* 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 $d_i = \begin{cases} 1 & si \quad y_i < 0 \\ -1 & autrement \end{cases}$

Au début du calcul, $x_0 = X$, $y_0 = Y$ et $z_0 = 0$. Après un nombre n suffisamment grand d'itérations, les variables x_n, y_n et z_n sont approximativement:

$$x_n \approx A_n \cdot \sqrt{X^2 + Y^2},$$
$$y_n \approx 0,$$
$$z_n \approx \tan^{-1}\left(\frac{Y}{X}\right).$$

Si on néglige la constante A_n, ces formules itératives permettent de calculer la longueur d'un vecteur et l'angle du vecteur avec l'axe *x*.

a. Supposez, par exemple, $x_0 = -24$, $y_0 = 32$ et $z_0 = 0$ au début. Calculez à la main la valeur de x_2 et y_2. Ne calculez pas z_2.

b. En utilisant les formules itératives ci dessus, écrivez une fonction MIPS pour calculer en 31 itérations la longueur et l'angle d'un vecteur dont les composantes *x* et *y* sont disponibles dans les registres **$a0** et **$a1**. Ces valeurs sont des nombres entiers signés encodés en complément à deux sur 32 bits. Un tableau de mots 32-bits dans la mémoire principale (pointé par le registre **$a2**) contient les coefficients $\tan^{-1}(2^{-i})$ déjà calculés (vous ne devez pas les obtenir). L'élément 0 du tableau est le coefficient de l'itération 0, l'élément 1 du tableau est le coefficient de l'itération 1, etc. Ignorez les dépassements de capacité possibles. Vous disposez des instructions

```
srav   rd, rs, rt
srlv   rd, rs, rt
sllv   rd, rs, rt
```

qui calculent en **rd** la valeur de **rs** décalée de **rt** positions à droite ou à gauche (**r** ou **l**) et de façon arithmétique ou logique (**a** ou **l**).

a)
On trouve x_2 et y_2, en utilisant les formules itératives. Au début, on a :

$x_0 = -24$
$y_0 = 32$
$z_0 = 0$

L'itération 1 donne:

$y_0 > 0 \Rightarrow d_0 = -1$
$x_1 = -24 - (-1)\cdot 32\cdot 2^0 = -24 + 32 = 8$
$y_1 = 32 + (-1)\cdot(-24)\cdot 2^0 = 32 + 24 = 56$

En suite, l'itération 2 donne:

$y_1 > 0 \Rightarrow d_1 = -1$
$x_2 = 8 - (-1)\cdot 56\cdot 2^{-1} = 8 + 28 = 36$
$y_2 = 56 + (-1)\cdot 8\cdot 2^{-1} = 56 - 4 = 52$

b)
On écrit une fonction MIPS pour calcule la longueur et l'angle d'un vecteur, en utilisant les formules « CORDIC ».

```
cordic:         add  $t0, $zero, $a0     ; t0 <- x0
                add  $t1, $zero, $a1     ; t1 <- y0
                add  $t2, $zero, $zero   ; t2 <- z0 (init to 0)
                add  $t3, $zero, $a2     ; t3 <- a2 (coef table)
                addi      $t4, $zero, 1  ; t4 <- 1 (index)
loop:           sltiu     $t5, $t4, 32   ; if t4 = 32
                beq  $t5, $zero, fin     ; then goto fin
                slt  $t5, $t1, $zero     ; if yi < 0
                bne  $t5, $zero, dpos    ; then goto dpos, else
dneg:           srav $t5, $t1, $t4       ; t5 <- yi*2**(-i)
                add  $t0, $t0, $t5       ; compute next xi
                srav $t5, $t0, $t4       ; t5 <- xi*2**(-i)
                sub  $t1, $t1, $t5       ; compute next yi
                lw   $t5, 0($t3)         ; t5 <- coeff[i]
                add  $t2, $t2, $t5       ; compute next zi
                j    cont                ; goto cont
```

```
dpos:       srav $t5, $t1, $t4      ; t5 <- yi*2**(-i)
            sub  $t0, $t0, $t5      ; compute next xi
            srav $t5, $t0, $t4      ; t5 <- xi*2**(-i)
            add  $t1, $t1, $t5      ; compute next yi
            lw   $t5, 0($t3)        ; t5 <- coeff[i]
            sub  $t2, $t2, $t5      ; compute next zi
cont:       addi $t3, $t3, 4        ; t3 <- next coeff addr
            addi $t4, $t4, 1        ; t4 <- next index
            j    loop               ; goto loop
fin:        add  $v0, $zero, $t0    ; v0 <- xn
            add  $v1, $zero, $t2    ; v1 <- zn
            jr   $ra                ; return
```

Une version plus compacte (avec moins d'instructions, car on utilise plus de registres temporaires) de la fonction est la suivante:

```
cordic:     add   $t0, $zero, $a0   ; t0 <- x0
            add   $t1, $zero, $a1   ; t1 <- y0
            add   $t2, $zero, $zero ; t2 <- z0 (init to 0)
            add   $t3, $zero, $a2   ; t3 <- a2 (coef table)
            addi  $t4, $zero, 1     ; t4 <- 1 (index)
loop:       sltiu $t5, $t4, 32      ; if t4 = 32
            beq   $t5, $zero, fin   ; then goto fin
            srav  $t6, $t1, $t4     ; t6 <- yi*2**(-i)
            srav  $t7, $t0, $t4     ; t7 <- xi*2**(-i)
            lw    $t8, 0($t3)       ; t8 <- coeff[i]
            slt   $t5, $t1, $zero   ; if yi < 0
            bne   $t5, $zero, dpos  ; then goto dpos, else
dneg:       add   $t0, $t0, $t6     ; compute next xi
            sub   $t1, $t1, $t7     ; compute next yi
            add   $t2, $t2, $t8     ; compute next zi
            j     cont              ; goto cont
dpos:       sub   $t0, $t0, $t5     ; compute next xi
            add   $t1, $t1, $t5     ; compute next yi
            sub   $t2, $t2, $t5     ; compute next zi
cont:       addi  $t3, $t3, 4       ; t3 <- next coeff addr
            addi  $t4, $t4, 1       ; t4 <- next index
            j     loop              ; goto loop
fin:        add   $v0, $zero, $t0   ; v0 <- xn
            add   $v1, $zero, $t2   ; v1 <- zn
            jr    $ra               ; return
```

On veut écrire une fonction permettant de calculer le reste de la division entière d'un nombre non signé de 32 bits par 15 sans effectuer la division même. Pour le calcul du reste, on utilise la propriété suivante : le reste de la division d'un nombre par 15 est la somme récursive des chiffres représentant ce nombre en base 16. On peut donc commencer par sommer tous les chiffres du nombre donné. Si la somme obtenue est représentée par plus qu'un seul chiffre hexadécimal (c'est-à-dire qu'elle est strictement supérieure à 15), alors on applique la même procédure récursivement: on somme les chiffres représentant la somme jusqu'a obtenir une valeur représentée par un seul chiffre hexadécimal. Enfin, si la somme finale obtenue est égale à 15, il faut remplacer le résultat par 0.

***Example**. Soit N = 0x3204'1EF2 = 839,130,866. On calcule le reste de la division par 15 comme suit. Tout d'abord, on somme les 8 chiffres hexadécimaux représentant N :*

3 + 2 + 0 + 4 + 1 + E + F + 2 = 0x29 = 41.

Comme la somme obtenue est représentée sur 2 chiffres hexadécimaux, on en calcule encore la somme :

2 + 9 = 0xB = 11.

*Cette valeur est le reste recherché, puisque la somme consiste en un seul chiffre hexadécimal et le résultat est différent de 0xF (donc pas besoin de le remplacer par 0). En effet, il est facile de vérifier que 839,130,866 = 55,942,057 * 15 + 11.*

a. Ecrivez une fonction qui suit les conventions MIPS et qui calcule le reste de la division d'un nombre *N* de 32 bits par 15 et retourne le reste. *N* est le seul paramètre fourni en entrée dans le registre **`$a0`**. La valeur du registre **`$a0`** ne doit pas forcement être préservée. Exploitez la structure du calcul en écrivant une fonction récursive.

b. Pour l'instruction ou les instructions qui additionnent les chiffres hexadécimaux, indiquez si vous utilisez **add** ou **addu**.
Expliquez en quoi changerait le comportement de ce programme particulier si l'on choisissait l'autre instruction.

a)
On fait les choix suivants :

- Pour isoler les 4 bits nécessaires au calcul des sommes partielles, on utilise successivement les instructions **`sllv`** et **`srlv`**. On aurait aussi pu utiliser un masque.
- Pour effectuer récursivement les sommes partielles, on peut utiliser l'instruction de saut « jump » (**`j`**) au lieu de « jump and link » (**`jal`**) puisque seul le registre **`$ra`** doit être preservé.

Un programme possible est décrit ci-dessous.

```
start:  addi $t0, $zero, 28
        addi $t1, $zero, 28
        add  $t2, $zero, $zero ; (somme intermédiaire)
        addi $t3, $zero, 8     ; (chiffres hex. dans 32 bits)

sum :   beq  $t3, $zero, rec   ; (test fin sommes partielles)
        sllv $t4, $a0, $t0
        srlv $t4, $t4, $t1
        add  $t2, $t2, $t4     ; (somme partielle)
        addi $t0, $t0, -4      ; (valeur shift à gauche)
        addi $t3, $t3, -1      ; (mise à jour compteur boucle)
        j    sum

rec :   add  $t5, $t2, $zero
        andi $t5, $t5, 0xFFF0
        beq  $t0, $zero, fin   ; (test de récursion)
        add  $a0, $t2, $zero   ; (update $a0)
        j    start             ; (récursion)

fin :   add $v0, $zero, $t2
        jr  $ra
```

Remarque : il n'est pas faux d'utiliser « jal » pour la récursion. Dans ce cas, il faut néanmoins sauvegarder l'adresse de retour **`$ra`** sur la pile, car chaque nouvel appel de procédure écrase la valeur stockée précédemment.

b)
On peut utiliser l'une ou l'autre de ces instructions. En effet, la seule différence entre **add** et **addu** consiste dans la génération du fanion d'overflow. Mais, la valeur stockée dans le registre cible est la même pour les 2 instructions. Enfin, on peut remarquer aussi que le calcul des sommes partielles ne peut de toute façon pas générer d'overflow puisque 0x8 * 0xF < 2^31.