POLYTECHNIQUE
MONTRÉAL

INF1500
Logique des systèmes numériques

Laboratoire 4

Soumis par:
Mohamad Awad - 2034584
Lourhmati Oussama - 2081643
Groupe 05

Le 19 novembre 2020

Introduction

Le système de ce laboratoire est un circuit en logique combinatoire conçu uniquement en utilisant le langage de description de hardware VHDL. Le circuit est composé de plusieurs modules qui seront décrit en détail dans la description du système: module MOD_3 (modulo 3), d'un module CMP (comparateur), d'un module MUX2_1 (multiplexeur 2 vers 1) et d'un module REG (registre). Le code VHDL de chaque module comporte 3 parties principales:

- **La librairie (library) :** contient les définitions de types et des fonctions spécifiques au VHDL. La librairie utilisée dans ce laboratoire est IEEE, elle comporte le fichier "STD_LOGIC_1164.ALL" qui contient les définitions dont nous avons besoin.

- **L'entité (entity) :** décrit l'interface du système avec le monde extérieur, c'est à dire les entrées (in) et les sorties (out) du système , déclarées dans port(...);

- **l'architecture (architecture):** décrit le comportement du système et qui est composée de deux parties :
    - ➢ **une partie déclarative :** qui déclare les signaux internes au module(s'il ya lieu)
    - ➢ **Le corps de l'architecture :** qui contient l'ensemble des énoncés décrivant le comportement du système entre les deux mots réservés **begin** et **end**

Une entité peut avoir plusieurs architectures différentes, ce qui n'est pas le cas dans ce laboratoire.

Toutes les entrées, sorties et signaux internes sont de type STD_LOGIC (ou STD_LOGIC_VECTOR pour les vecteurs à plusieurs bits).

Dans le cadre de ce laboratoire, au lieu d'utiliser les portes logiques nous avons opté pour les instructions *with/select* et *when/else* pour la description en flot de données.

Description du système

Module MOD_3

Le module **MOD_3** effectue l'opération modulo 3 sur une entrée A de type STD_LOGIC_VECTOR , “(4 downto 0)” signifie un vecteur d’entrée à 5 bits. La sortie est également STD_LOGIC_VECTOR (2 downto 0) : vecteur de sortie à 3 bits qui contient le résultat.

Le corps de l’architecture **MOD_3Behave** est divisé en trois sections: lorsque S prend un des seuls trois résultats possibles de l'opération modulo 3: "000", "001" et "010", (0,1 et 2), utilisant “with/select”. Le code prend en considération toute valeur de 5 bits possible (0 à 31 en décimal) (pas d'équation). Au lieu de mettre les possibilités cas par cas, nous les regroupons dans les trois sections que nous décrivons afin de rendre le code mieux structuré.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity MOD_3 is
    port(
        A: in STD_LOGIC_VECTOR(4 downto 0);
        S: out STD_LOGIC_VECTOR(2 downto 0)
    );
end MOD_3;

architecture MOD_3Behave of MOD_3 is
begin
    with A select S <=
"000" when "00000" | "00011" | "00110" | "01001" | "01100" | "01111"|
"10010" | "10101" | "11000" | "11011" | "11110" ,
"001" when "00001" | "00100" | "00111" | "01010" | "01101" | "10000"|
"10011" | "10110" | "11001" | "11100" | "11111" ,
"010" when others;
end MOD_3Behave;
```

Module CMP

Le module CMP est un comparateur : il compare ses deux entrées A et B : in STD_LOGIC_VECTOR (4 downto 0) , et a comme sortie S: out STD_LOGIC_VECTOR (2 downto 0). L'architecture **CMPBehavioral** est basée sur "when/else". Nous utilisons les opérateurs **=** et **<**. Ainsi, lorsque A= B, la sortie S (sur 3 bits) est **"001"**; lorsque A < B, la sortie S est **"010";** et finalement, si ce n'est pas égal ou inférieur, alors c'est forcément supérieur (sortie S est **"100"**), nous le mettons directement après le dernier **else .**Il s'agit d'une bonne pratique en programmation :un cas par défaut. Le code VHDL de ce module correspond au fichier **CMP.vhd** dans le projet vivado.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;



entity CMP is
 port(
        A: in STD_LOGIC_VECTOR(4 downto 0);
        B: in STD_LOGIC_VECTOR(4 downto 0);
        S: out STD_LOGIC_VECTOR(2 downto 0)
    );
end CMP;

architecture CMPBehavioral of CMP is

begin
        S <= "001" when A = B else
             "010" when A < B else
             "100" ;

end CMPBehavioral;
```

Module MUX2_1

Le module MUX2_1 : multiplexeur qui sélectionne entre ses entrées A et B : in STD_LOGIC_VECTOR (2 downto 0), en fonction du signal de sélection SEL : in STD_LOGIC. Également la sortie S est à 3 bits : out **STD_LOGIC_VECTOR (2 downto 0)**. Dans l'architecture **MUX2_1Behavioral,** nous codons le comportement du module en quelques lignes seulement:

- S prend la valeur de A lorsque SEL = '0',
- S prend la valeur de B lorsque SEL = '1'.

Le code VHDL de ce module correspond au fichier **MUX2_1.vhd** dans le projet vivado.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity MUX2_1 is
    port(
            A: in STD_LOGIC_VECTOR(2 downto 0);
            B: in STD_LOGIC_VECTOR(2 downto 0);
            SEL: in STD_LOGIC;
            S: out STD_LOGIC_VECTOR(2 downto 0)
    );
end MUX2_1;

architecture MUX2_1Behavioral of MUX2_1 is

begin
        S <= A when SEL = '0' else
                B when SEL = '1';
end MUX2_1Behavioral;
```

Module REG

Le module REG (registre) est un loquet à trois bits , il reçoit une entrée D(in) et a une sortie S(out) : vecteur de 3 bits: **STD_LOGIC_VECTOR (2 downto 0)** et se comporte dépendamment de son entrée E(enable): **in STD_LOGIC.**

- Si E=1. La sortie du REG prend la valeur de son entrée D.
- Si E=0. La valeur de la sortie du REG est gardée, indépendamment de son entrée D, à la dernière valeur stockée avant que E change de 1 à 0.

Dans la partie déclarative de l'architecture **REGBehavioral** , nous déclarons un signal interne

(vecteur de 3 bits) : **STD_LOGIC_VECTOR (2 downto 0) ,** en tant que variable temporaire. Le signal T sauvegarde la valeur de l'entrée D lorsque le E (enable) est à 1. Et si E change à 0, la sortie S prend la valeur enregistrée dans T. Nous utilisons "when/else" dans le code.

Le code VHDL de ce module correspond au fichier **REG.vhd** dans le projet vivado.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity REG is
    port(
        D: in STD_LOGIC_VECTOR(2 downto 0);
        E: in STD_LOGIC;
        S: out STD_LOGIC_VECTOR(2 downto 0)
    );
end REG;


architecture REGBehavioral of REG is
    signal T: STD_LOGIC_VECTOR(2 downto 0);
begin
    T <= D when E = '1';
    S <= D when E = '1' else
         T when E = '0';
end REGBehavioral;
```

Système global

Description structurale

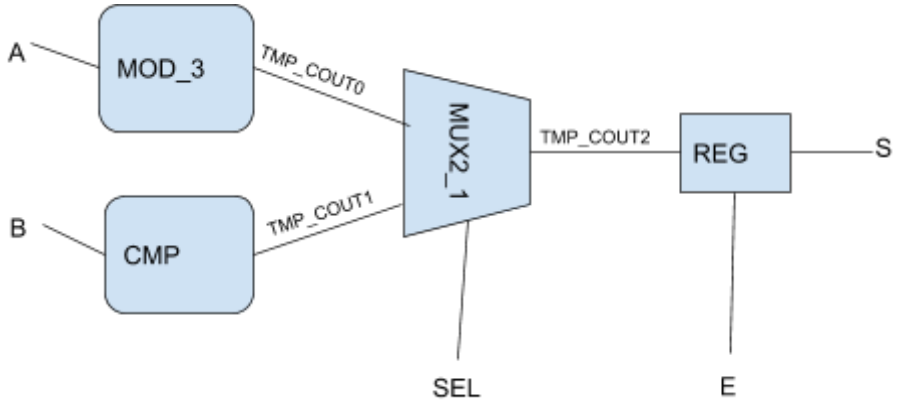
Le code VHDL est obtenu en définissant une entité propre au système global : **entity SYSTEM ,** nous déclarons les entrées et sorties propres au système dans port(..);

- A: sera l'entrée des modules **MOD_3 et CMP** (vecteur à 5 bits).
- B : sera la deuxième entrée du module **CMP** (vecteur à 5 bits).
- E : sera l'entrée du module **REG.**
- S : sera l'entrée du module **MUX2_1.**

Dans la partie déclarative de l'architecture appelée **SYSTEMBehavioral**

- on déclare les différents modules de la même façon qu'on déclare leurs **entités ,** mais en remplaçant le mot **entity** par le mot **component**.
- Les signaux internes au système : **TMP_COUT0 , TMP_COUT1 et TMP_COUT2** qui sont des vecteurs à 3 bits.

Dans le corps de l'architecture, quatres énoncés concurrents **U0,U1,U2,U3** vont instancier les différents modules (**components**) et **portmap** spécifie les interconnexions dans chacuns de ces derniers:



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity SYSTEM is
    port(
        A: in STD_LOGIC_VECTOR(4 downto 0);
        B: in STD_LOGIC_VECTOR(4 downto 0);
        E: in STD_LOGIC;
        SEL: in STD_LOGIC;
        S: out STD_LOGIC_VECTOR(2 downto 0)
    );
end SYSTEM;

architecture SYSTEMBehavioral of SYSTEM is
    --MOD3--
    component MOD_3 is
        port(
            A: in STD_LOGIC_VECTOR(4 downto 0);
            S: out STD_LOGIC_VECTOR(2 downto 0)
        );
    end component;

    --CMP--
    component CMP is
        port(
            A: in STD_LOGIC_VECTOR(4 downto 0);
            B: in STD_LOGIC_VECTOR(4 downto 0);
            S: out STD_LOGIC_VECTOR(2 downto 0)
        );
    end component;

    --MUX2_1--
    component MUX2_1 is
        port(
            A: in STD_LOGIC_VECTOR(2 downto 0);
            B: in STD_LOGIC_VECTOR(2 downto 0);
            SEL: in STD_LOGIC;
            S: out STD_LOGIC_VECTOR(2 downto 0)
        );
    end component;
```

```
 --REG--
    component REG is
        port(
            D: in STD_LOGIC_VECTOR(2 downto 0);
            E: in STD_LOGIC;
            S: out STD_LOGIC_VECTOR(2 downto 0)
        );
    end component;

    signal TMP_COUT0: STD_LOGIC_VECTOR(2 downto 0);
    signal TMP_COUT1: STD_LOGIC_VECTOR(2 downto 0);
    signal TMP_COUT2: STD_LOGIC_VECTOR(2 downto 0);

begin
    U0 :
        MOD_3 port map(
        A => A, S => TMP_COUT0
        );
    U1:
        CMP port map(
        A => A, B => B, S => TMP_COUT1
        );
    U2:
        MUX2_1 port map(
        A => TMP_COUT0, B => TMP_COUT1, SEL => SEL, S => TMP_COUT2
        );
    U3:
        REG port map(
        E => E, D => TMP_COUT2, S => S
        );
end SYSTEMBehavioral;
```

Vérification du système

MOD_3

Pour la simulation du MOD_3 :

- On applique un force clock à touts les bits d'entrées du vecteur d'entrée A.
    - A(0) à 1 ns
    - A(1) à 2 ns
    - A(2) à 4 ns
    - A(3) à 8 ns
    - A(4) à 16 ns
    - La simulation est faite sur un intervalle de 16 ns pour couvrir toutes les périodes et assurer toutes les 32 combinaisons possibles.
- Le but du force clock est de tester toutes les entrées possibles puisque le code VHDL de ce module décrit le flot de données et compte toute les possibilités sans aucune équation ou formule pour faire l'opération
- La Figure 2.0 valide le fonctionnement du module, et donne un résultat attendu pour chaque entrée
    - Exemple: 9%3 =0 , le reste de la division de 9 par 3. (ligne jaune).

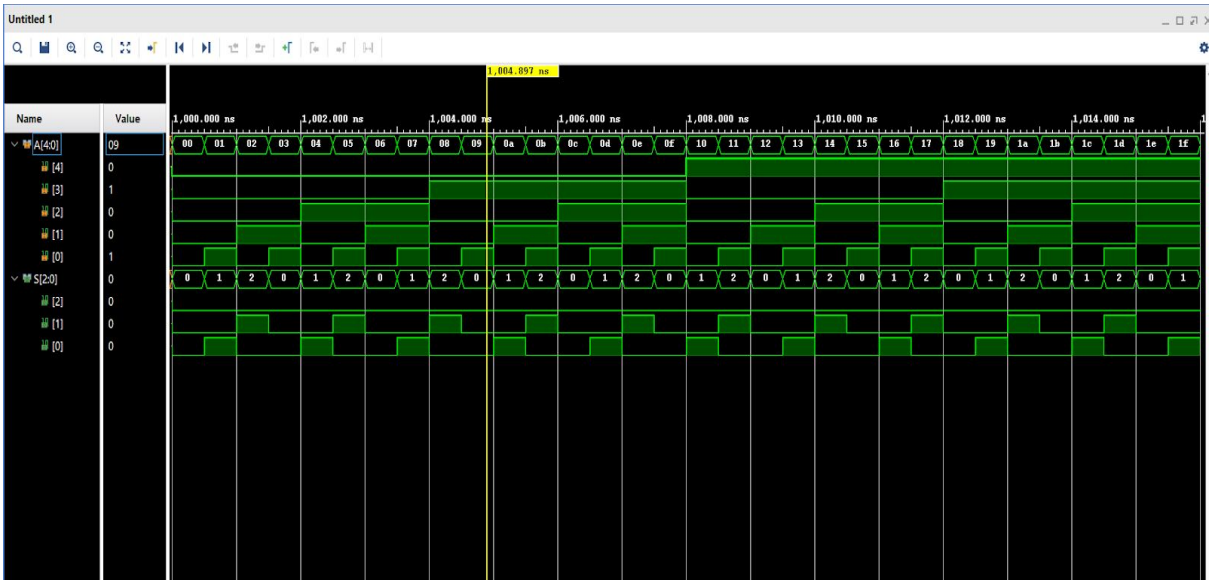


Figure 2.0: Simulation Modulo 3

CMP

Pour tester le module CMP. Nous procédons de la façon suivante

- force clock pour A(0) et A(1)
    - A(0) à 1 ns
    - A(1) à 2 ns
    - La simulation est faite donc sur 2 ns
- force constant pour le reste des bits de A : A(2)= A(3)= A(4) =0
- force constant pour le vecteur B : B = 2(hex) = 00010

De cette manière on compare techniquement entre deux nombre de deux bits , un qui est toujours 2 et l'autre qui change à une valeur entre 0 et 3 chaque 0.5 ns.

Le but est de simplifier le test du comparateur et vérifier que le code VHDL est bon.

La figure 2.1 valide le bon fonctionnement et un résultat attendu pour chaque cas, comme demandé dans l'énoncé :

- Pour A= 00010 . A=B. S=001
- Pour A =00000 et A =00001 . A<B et S= 010
- Pour A=00011. A>B et S = 100.

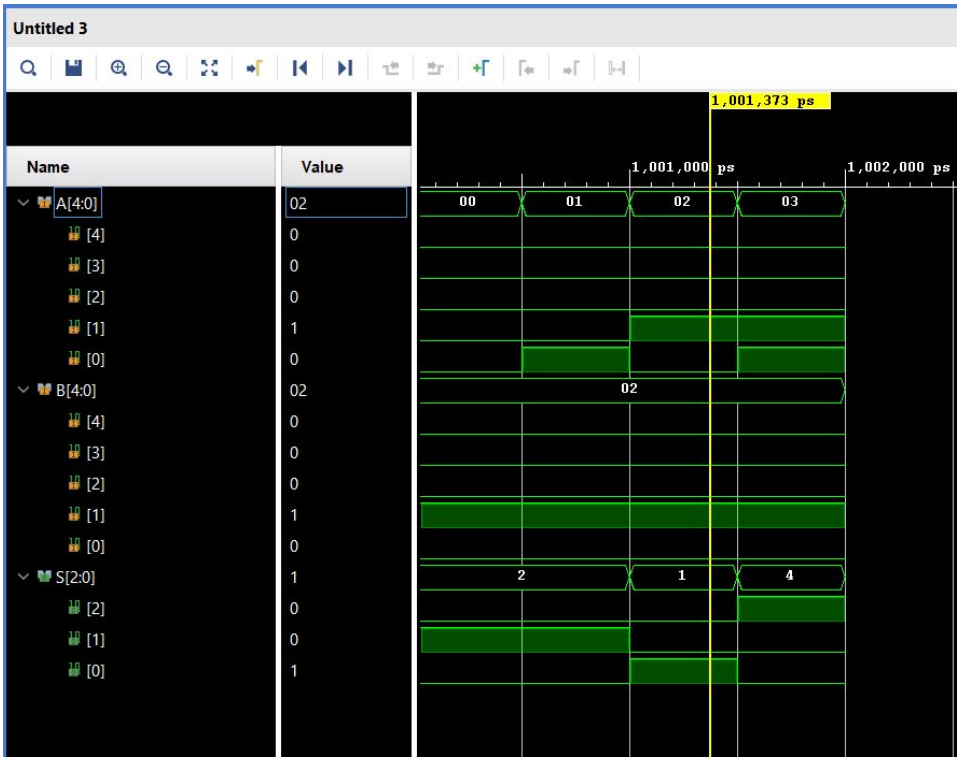


Figure 2.1: Simulation CMP

MUX2_1

Pour tester le module MUX2_1 , nous procédons de la façon suivante:

- Force constant pour le vecteur A. A =001.
- Force constant pour le vecteur B. B=010.
- Force clock pour SEL : période : 50 ns.

Le But est de tester le comportement du multiplexeur en fonction du signal de sélection,

donc ce n'est pas nécessaire de varier les entrées. Juste le signal de sélection qui change chaque 25 ns. (Une seule fois ici puisque la simulation est faite sur 50 ns).

La Figure 2.2 valide le bon fonctionnement du système comme demande l'énoncé.

- ❖ Pour SEL =0 . S = A = 001.
- ❖ Pour SEL =1 . S =B = 010.

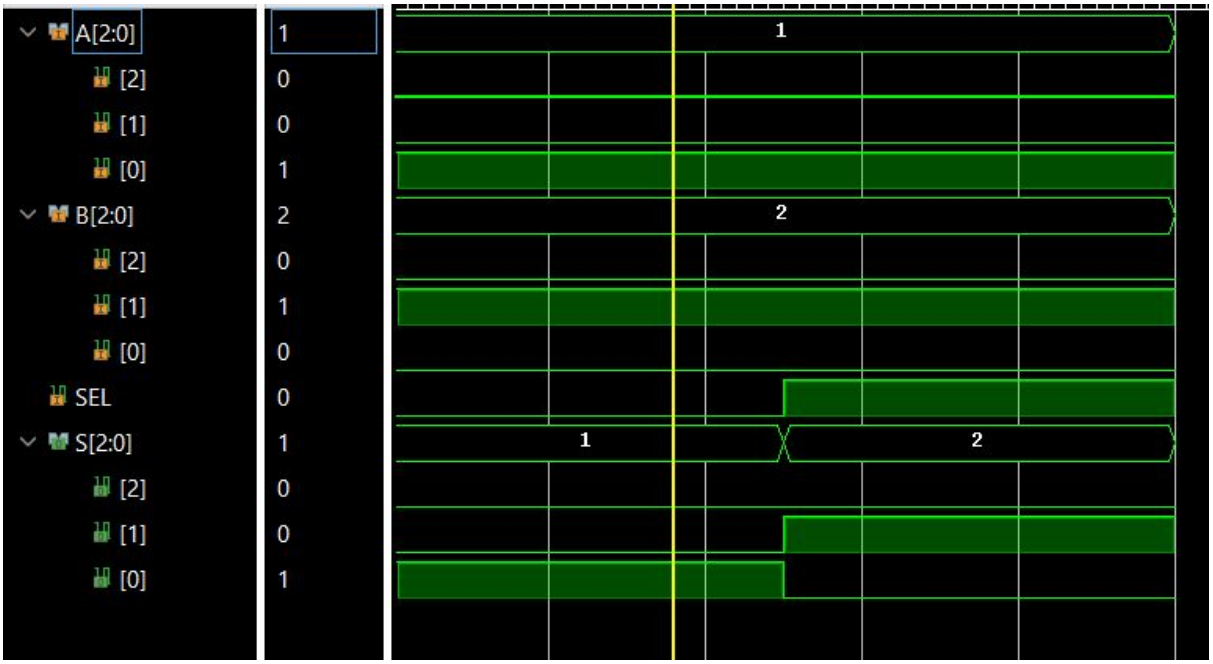


Figure 2.2 : Simulation MUX2_1

REG

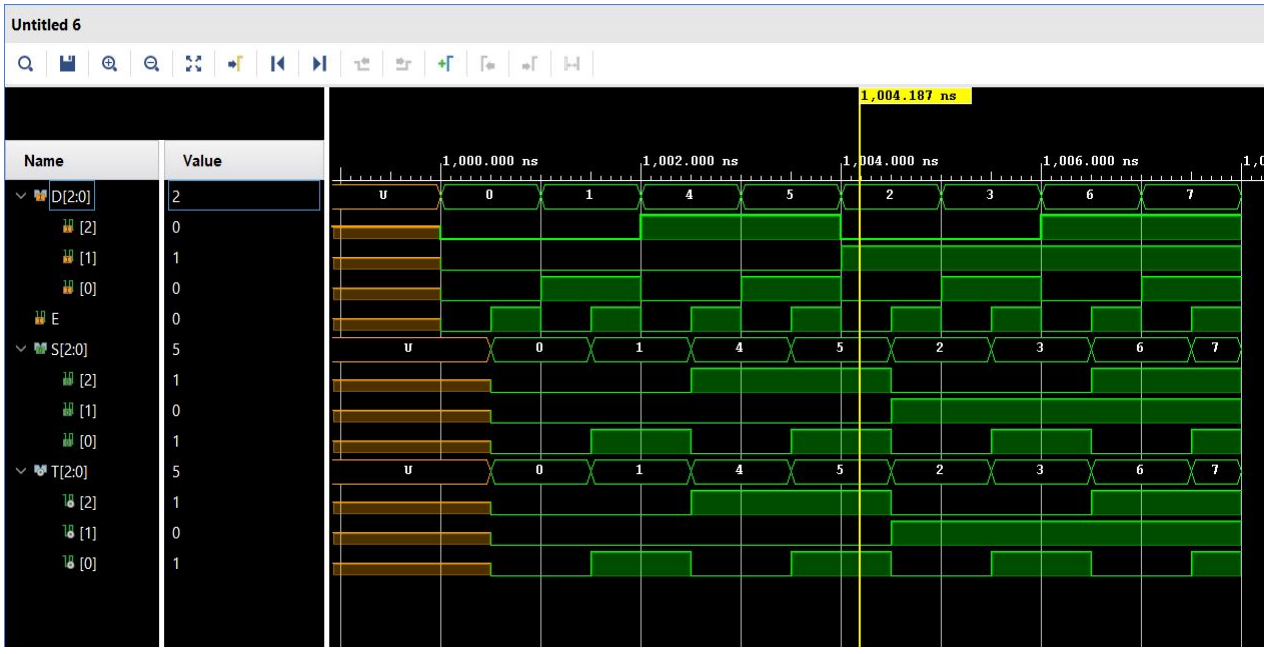
Pour tester le module REG, nous procédons de la façon suivante :

- force clock à E (enable) de période 1 ns.
- force clock à chaque bit de D:
  - D(0) à 2 ns.
  - D(1) à 8 ns
  - D(2) à 4 ns.
- La simulation est faite sur un intervalle de 8 ns.

Le but de donner des périodes dans un ordre aléatoire et non croissant au bits de D et de mélanger les valeurs de D et ne pas les avoir en ordre croissant sur l'écran, pour rendre le test plus pertinent et montrer que S bloque la valeur de D une fois E change à 0. De plus, E ne doit pas prendre la plus grande période de simulation, pour éviter un cas où E change à 0 toujours avec une valeur de D qui est la même que celle de la dernière fois.

La figure 2.3 valide le bon fonctionnement du système.

Nous pouvons remarquer également que S prenait la valeur U orange, au tout début de la simulation, puisque D avait cette valeur avant le début, et au début E était à 0, donc S prend la valeur la plus récente de D. Aussi, autour de la ligne jaune. nous pouvons remarquer que entre la 4ème et la 5ème ns , S=5 lorsque E=0, elle conserve la valeur la plus récente de D lorsque E était 1 (entre 3.5 et 4 ns). et S prend la valeur de D actuelle (2) lorsque E revient à 1 (à 4.5 ns)



.**Figure 2.3 : Simulation REG**

SYSTEM.

Finalement, pour tester le système globale, il suffirait de changer les valeurs de SEL et E seulement, puisque tous les modules ont été testés rigoureusement, donc

- Force clock à E (25 ns)
- Force clock à SEL (50 NS)
- Force constant à A. A= 0a = 01010
- Force constant à B. B= 0b = 01011

Puisque L'entrée D du registre est la sortie du multiplexeur, la sortie du système global dépend de SEL seulement lorsque E=1 et bloque la valeur de D lorsque E=0 , indépendamment de SEL.

Ceci est bien validé dans la figure 2.4, simulation sur un intervalle de 50 ns.

Par Exemple à la ligne jaune, E=0, et SEL =1, donc la sortie concerve la valeur la plus recente de D lorsque E était à 1, (il coincide que SEL était à 0). La sortie S est donc la sortie du multiplexeur pour SEL=0 : 0a % 3 = 10%3 = 1 =001 .

Ensuite lorsque E revient à 1 , il coïncide que SEL est à 1 (à 37.5 ns) donc la sortie prend la valeur actuelle de D, sortie du multiplexeur pour SEL =1 et qui compare A et B:

A= 0a < B=0b donc la sortie du comparateur est 010 , ce qui est bien la valeur de S à partir de 37.5 ns

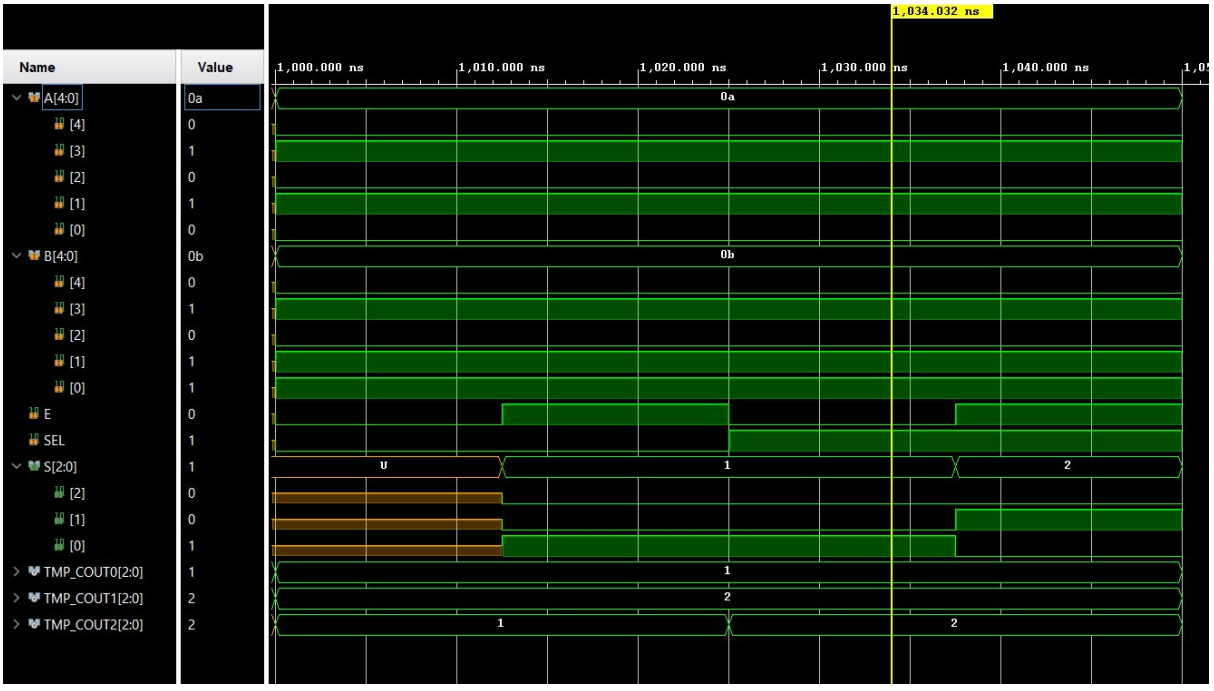


Figure 2.4 : Simulation du SYSTEM.

Conclusion

Pour conclure, notre circuit logique est composé de 4 modules que nous décrivons en détail dans la description du système. Les modules sont le MOD_3, le comparateur CMP, le multiplexeur MUX2_1 et le registre REG. Pour les concevoir, nous utilisons uniquement le code VHDL. Au lieu d'utiliser les portes logiques, nous nous servons uniquement des instructions *with/select* et *when/else* pour la description en flot de données. Par la suite, pour valider le système nous élaborons des simulations pour chaque module, tel qu'expliqué dans la vérification du système. Bref, de cette manière, il est clair que le système fonctionne correctement et qu'il a le comportement désiré.