COMPTE-RENDU SYSTÈMES SUR PUCES

« GCD »

Réalisé par Sean Marotta & Pierre-Élie Morrot

Enseignant : Julien Dubois & Barthélémy Heyrman
Année : 2020 – 2021

➢ INTRODUCTION :

Dans ce TP, nous allons implémenter un FPGA qui aura pour fonction de calculer le PGCD entre deux nombres x et y. Le projet sera codé en VHDL grâce au logiciel Vivado de Xilinx. L'algorithme qui permet de calculer ce PGCD est le suivant :

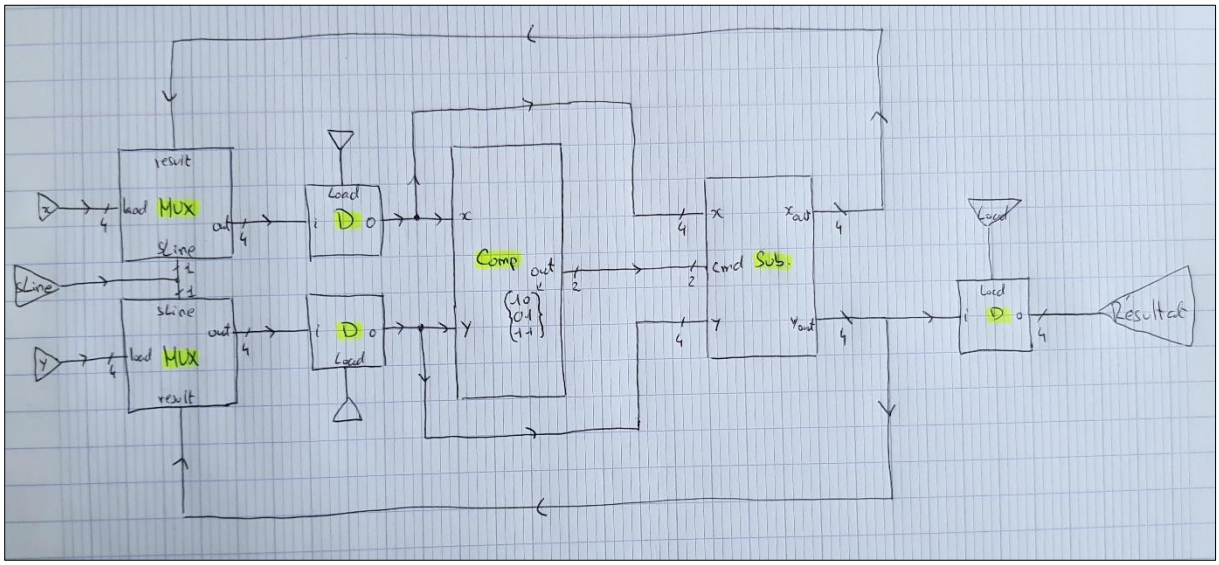
```
x = 10
y=2
Is x = y? No, x > y therefore x = x - y
in our case, x = 10 - 2 = 8.
Is x = y? No, x > y therefore x = x - y
in our case, x = 8 - 2 = 6.
Is x = y? No, x > y there fore x = x - y
in our case, x = 6 - 2 = 4.
Is x = y? No, x > y therefore x = x - y
in our case, x = 4 - 2 = 2.
Is x = y? Yes, therefore the GCD of 10 and 2 is 2.
```

Pour cela, nous avons à notre disposition plusieurs composants déjà implémentés qui sont :

- Mux: takes 2 of 4-bit inputs and one select line. Based on the select line, it outputs either the 1st 4-bit number or the 2nd 4-bit number.
- Reg: Takes a 4-bit input, a load signal, reset, and a clock signal. If the load signal is high and the clock is pulsed, it outputs the 4-bit number.
- Comparator: Takes 2 of 4-bit numbers, and assets one of 3 signals depending on whether the 1st number is less than, greater than or equal to the 2nd number.
- Subtractor: Takes 2 of 4-bit numbers, subtracts the smaller number from the larger.
- Output Reg: Holds the GCD value. When x = y the GCD has been found and can be outputted. Because it is a register entity it should also take a clock and reset signal.

➢ PREMIÈRE ÉTAPE – SCHÉMA DE CÂBLAGE :

La première étape a été de dessiner le schéma de câblage sur papier (quels composants utiliser, faire apparaître les entrées et les sorties etc…). Voici donc le résultat obtenu (sans *clock* et *reset*) :

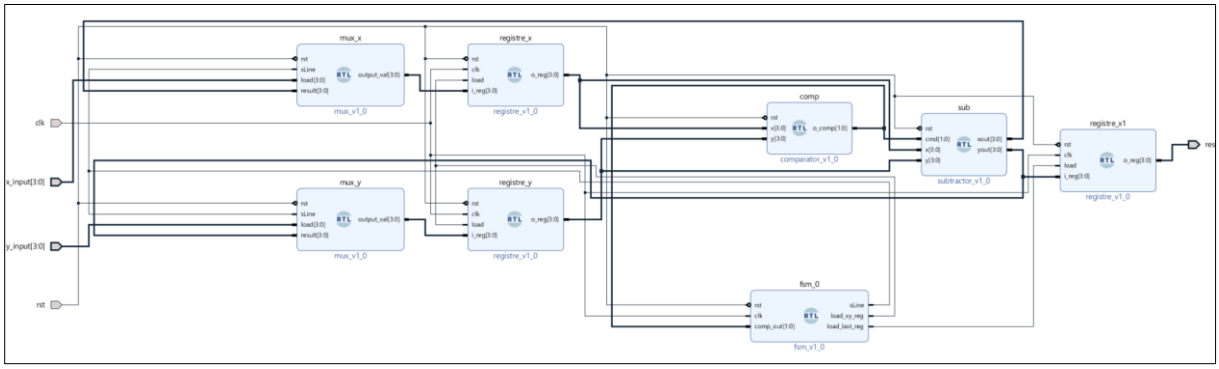


➢ DEUXIÈME ÉTAPE – BLOCK DESIGN :

Une fois le schéma de câblage établi, nous pouvons faire le diagramme sur Vivado en fonction de celui-ci. Pour cela :

1. Création d'un Block Design
2. Importation des différents composants
3. Placer ces composants dans le diagramme
4. Établir les connexions entre les composants
5. Ajout des entrées et sorties (clic droit → Create Port)

Voici le résultat :



Un composant FSM apparaît, c'est la « Finite State Machine » ou machine à état que nous allons expliquer maintenant.

➢ TROISIÈME ÉTAPE – FSM :

Ce composant FSM a pour rôle de gérer automatiquement l'état des différentes entrées qui sont dans notre cas :

- *sLine* : bit de sélection des entrées des deux MUX (premier coup à 0 et ensuite toujours à 1).
- *load_xy_reg* : bit de chargement des registres pour les entrées x et y (toujours à 1).
- *load_last_reg* : bit de chargement du registre de sortie (à 1 lorsque $x = y$).

On peut donc établir trois états :

	STATE_0	STATE_1	STATE_2
load_last_reg	0	0	1
load_xy_reg	1	1	1
sLine	0	1	1

Voici le code correspondant au composant FSM :

```
entity fsm is
    Port ( rst : in STD_LOGIC;
           clk : in STD_LOGIC;
           comp_out : in STD_LOGIC_VECTOR (1 downto 0);
           sLine : out STD_LOGIC; -- toujours à 1 sauf au début
           load_xy_reg : out STD_LOGIC; -- always reg_x et reg_y (alway ON)
           load_last_reg : out STD_LOGIC); -- reg_last
end fsm;

architecture Behavioral of fsm is
constant STATE_0 : std_logic_vector(2 downto 0) := "010"; -- load_last_reg load_xy_reg sLine
constant STATE_1 : std_logic_vector(2 downto 0) := "011";
constant STATE_2 : std_logic_vector(2 downto 0) := "111";

SIGNAL CURRENT_STATE : std_logic_vector(2 downto 0);

begin
process(CLK)
begin
    if(CLK'event AND CLK='1')then
        case CURRENT_STATE is
            when STATE_0 =>
                CURRENT_STATE <= STATE_1;
            when STATE_1 =>
                if(COMP_OUT="11")then
                    CURRENT_STATE <= STATE_2;
                end if;
            when STATE_2 =>
                CURRENT_STATE <= STATE_0;
            when others =>
                CURRENT_STATE <= STATE_0;
        end case;
    end if;
end process;

SLine <= CURRENT_STATE(0);
load_xy_reg <= CURRENT_STATE(1);
load_last_reg <= CURRENT_STATE(2);

end Behavioral;
```

!!! Une fois toutes les étapes précédentes terminées, il faut générer le Wrapper HDL à faire qu'une seule fois (Clic droit → Create HDL Wrapper) et aussi générer le Block Design à chaque fois que l'on change quelque chose dans le diagramme (Generate Block Design). !!!

➢ QUATRIÈME ÉTAPE – TESTBENCH :

Dans cette étape nous allons, vérifier le bon fonctionnement du système que l'on a créé. Pour cela, nous allons créer le TB pour le tester (Sources → Clic Droit sur Simulation Sources → Add Sources → Add or create simulation sources → Create File → Finish).

Dans ce fichier, nous allons entrer manuellement l'états des entrées restants (*x_input*, *y_input*, *clk*, *rst*) et nous allons regarder ce qu'il y a en sortie. Voici le code correspondant au TB :

```
entity TB_gcd is
--  Port ( );
end TB_gcd;

architecture Behavioral of TB_gcd is
component design_1_wrapper is
    Port ( x_input : in STD_LOGIC_VECTOR (3 downto 0); -- Tous les ports qui corresponds aux Diagram du début
           y_input : in STD_LOGIC_VECTOR (3 downto 0);
           resultat: out STD_LOGIC_VECTOR (3 downto 0);
           clk,rst : in STD_LOGIC);
end component;

SIGNAL x,y, resultat : std_logic_vector(3 downto 0);
SIGNAL clk,rst : std_logic;

begin

UUT : design_1_wrapper PORT MAP (clk=>clk,resultat=>resultat,rst=>rst,x_input=>x,y_input=>y);

process
begin
    x <= "0110";
    y <= "1001";
    wait;
end process;

process
begin
    clk <= '0';
    wait for 20ns;
    clk <= '1';
    wait for 20ns;
end process;

end Behavioral;
```

➢ CINQUIÈME ÉTAPE – SIMULATION :

Dans cette étape, nous allons pouvoir visualiser l'état des signaux de notre système. Nous avons mis en entrée ***x = 0110 = 6*** et ***y = 1001 = 9***. Le GCD attendu est de ***3*** et c'est bien le résultat obtenu sur la sortie.