JUNIA ISEN

FPGA – TP2

Composants séquentiels

I - Exercices

Créer un nouveau projet « TP2_registres »

I.a Synthèse d'une bascule JK

Ce premier exercice vise à vous familiariser avec la description de procédés séquentiels Vous apprendrez à décrire des procédés **synchrones** (préférables), mais aussi **asynchrones**.

Vous pourrez vous aider de l'implémentation de bascule D décrite en annexe.

I.a.1 Bascule JK, partie synchrone

Le bloc réalisé dans cet exercice peut être décrit comme suit :

Bascule JK			
	Nom	**Taille**	**Description**
Entrées	J	1 bit	Entrée J
	K	1 bit	Entrée K
	CLK	1 bit	Horloge (sensibilité au front montant)
Sorties	Q	1 bit	Sortie de la bascule
	Qn	1 bit	Sortie complémentée de la bascule

Travail demandé :

1. Ajoutez à votre projet un nouveau fichier flipflop_JK.vhd
2. Rappelez le fonctionnement de la bascule JK (sous forme de table caractéristique par exemple)
3. Décrivez dans une *entity* nommée flipflop_JK le fonctionnement de la bascule
4. Ajoutez à votre projet un nouveau fichier de *testbench* tb_flipflop_JK.vhd, dans lequel vous décrirez les entrées à appliquer à la bascule pour tester son bon fonctionnement
5. Synthétisez et simulez votre bascule

I.a.2 Bascule JK, partie asynchrone

On souhaite ajouter à la bascule précédemment réalisée des entrées de mise à 0 et de mise à 1 **asynchrones**.

La description de la bascule JK devient :

Bascule JK			
	Nom	**Taille**	**Description**
Entrées	J	1 bit	Entrée J
	K	1 bit	Entrée K
	CLK	1 bit	Horloge (sensibilité au front montant)
	SETn	1 bit	Preset* (asynchrone, active à l'état bas)
	RSTn	1 bit	Reset* (asynchrone, active à l'état bas)
Sorties	Q	1 bit	Sortie de la bascule
	Qn	1 bit	Sortie complémentée de la bascule

Travail demandé :

1. Ajoutez à votre projet un nouveau fichier flipflop_JKrs.vhd
2. Rappelez le fonctionnement de la bascule JK avec entrées asynchrones de preset et reset.
3. Décrivez dans une *entity* nommée flipflop_JKrs le fonctionnement de la bascule
4. Ajoutez à votre projet un nouveau fichier de *testbench* tb_flipflop_JKrs.vhd, dans lequel vous décrirez les entrées à appliquer à la bascule pour tester son bon fonctionnement
5. Synthétisez et simulez votre bascule

I.b Synthèse d'un registre à décalage

Dans ce deuxième exercice, vous réaliserez l'implémentation de registres à décalage. Ici, il n'est pas demandé de réutiliser la bascule JK réalisée précédemment : il est possible de décrire des implémentations très compactes et plus lisibles qu'en instanciant des composants élémentaires.

Les registres à décalage réalisés auront une largeur fixe de **8 bits**.

I.b.1 Registre SISO (Serial In, Serial Out)

On décrit le registre à décalage 8b suivant :

Registre à décalage SISO			
	Nom	**Taille**	**Description**
Entrées	Si	1 bit	Serial in
	CLK	1 bit	Horloge (sensibilité au front montant)
	SETn	1 bit	Preset* (asynchrone, active à l'état bas)
	RSTn	1 bit	Reset* (asynchrone, active à l'état bas)
Sorties	So	1 bit	Sortie du registre

Travail demandé :

1. Ajoutez à votre projet un nouveau fichier shift_register_SISO8.vhd
2. Rappelez le fonctionnement d'un registre à décalage
3. Décrivez dans une *entity* nommée shift_register_SISO8 le fonctionnement du registre
4. Ajoutez à votre projet un nouveau fichier de *testbench* tb_shift_register_SISO8.vhd, dans lequel vous décrirez les entrées à appliquer au registre pour tester son bon fonctionnement
5. Synthétisez et simulez votre registre à décalage

I.b.2 Registre Universel

Dans l'exercice précédent, vous avez décrit un registre n'ayant qu'un mode opératoire (décalage à droite). Dans cet exercice, on ajoutera une qualité multi-opératoire comprenant toutes les opérations utiles des registres (Chargement/lecture série ou parallèle, décalage/rotation à gauche ou à droite).

Ces registres sont dits « universels ». Un exemple de registre universel 4 bits à 4 modes opératoires (Mémorisation, décalage à droite, décalage à gauche, chargement parallèle) est donné ci-dessous :

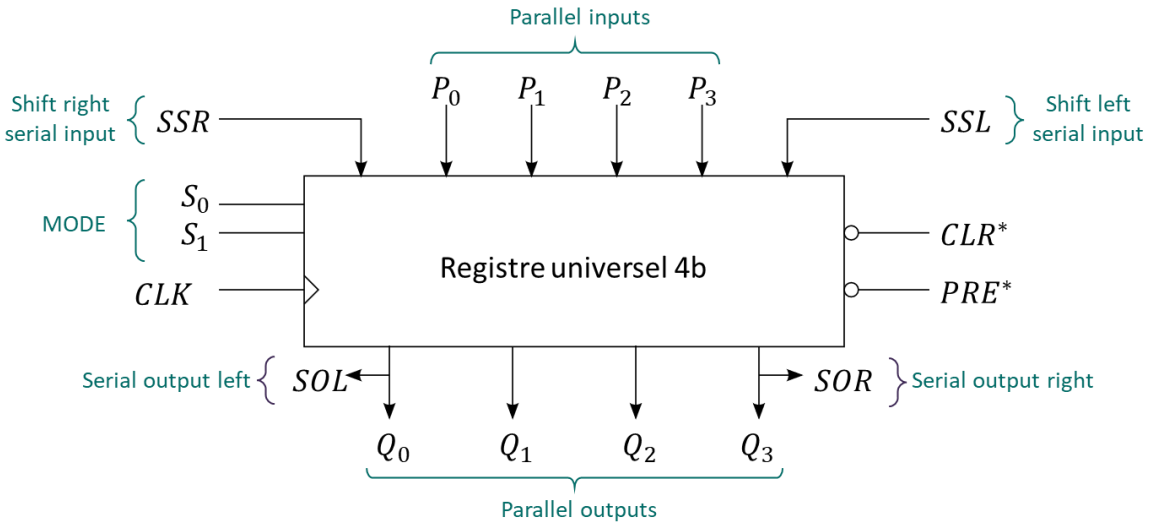

Figure 1: Exemple de registre universel 4 bits

Ici, on se propose d'implémenter un registre universel capable des six opérations suivantes, sélectionnables grâce à une entrée de sélection :

- Mémorisation
- Chargement parallèle
- Décalage à gauche
- Décalage à droite
- Rotation à gauche
- Rotation à droite

Registre universel			
	Nom	**Taille**	**Description**
Entrées	SSR	1 bit	Shift right serial input
	SSL	1 bit	Shift left serial input
	Pi	8 bits	Parallel input
	SEL	3 bits	Mode selection : - X00 : Hold (Memorize) - X11 : Parallel load - 001 : Shift right - 010 : Shift left - 101 : Rotate right - 110 : Rotate left
	CLK	1 bit	Horloge (sensibilité au front montant)
	SETn	1 bit	Preset* (asynchrone, active à l'état bas)
	RSTn	1 bit	Reset* (asynchrone, active à l'état bas)
Sorties	SOR	1 bit	Shift output right
	SOL	1 bit	Shift output left
	Qo	8 bits	Parallel outputs

Travail demandé :

1. Ajoutez à votre projet un nouveau fichier shift_register_universal8.vhd
2. Décrivez dans une *entity* nommée shift_register_universal8 le fonctionnement du registre
3. Ajoutez à votre projet un nouveau fichier de *testbench* tb_shift_register_universal8.vhd, dans lequel vous décrirez les entrées à appliquer au registre pour tester son bon fonctionnement
4. Synthétisez et simulez votre registre à décalage

Vous pouvez enfin tester votre registre universel directement sur la carte de développement :

Travail demandé :

1. Dans un nouveau fichier *toplevel.vhd*, instanciez votre registre universel et connectez-le aux entrées/sorties de la carte FPGA pour le faire fonctionner :

SSR	SW9
$P_7 - P_0$	0 (chargement parallèle inutilisé)
SSL	SW8
SEL	SW2-0
$RSTn$	KEY3[1]
$SETn$	KEY2
CLK	not(KEY0)
SOL	-
$Q_7 - Q_0$	LEDG7-0
SOR	-

2. Synthétisez et implémentez votre design sur la carte FPGA ; vérifiez le bon fonctionnement de votre registre universel.

[1] Les entrées KEY sont déjà actives à l'état bas, voir documentation de la carte Cyclone V GX Starter kit p. 29

I.b.3 Pour aller plus loin … Les *generic* (optionnel)

Plutôt que de décrire des composants de « taille fixe » (ex. registre 8 bits), il existe une syntaxe permettant de **paramétrer** les dimensions à l'aide d'une directive *generic*. Ceci permet à la place de décrire un comportement valable pour, par exemple, un registre N bits, puis de l'instancier avec la dimension requise, spécifique à l'application.

Souvent, cela permettra :

1) De décrire de façon générique des composants pour les réutiliser dans différents contextes sans développement supplémentaire (ex. utilisation conjointe de registres universel 4b, 8b, 16b, 32b, etc. avec une description unique)
2) De décrire de façon générique des composants, de les tester en dimensions réduites, puis de les déployer à taille réelle (ex. tester un registre universel 4b, puis le déployer à 64b avec une description unique)

Les paramètres se déclarent dans le champ *entity*, de la même manière que l'instruction *port*. Ces paramètres peuvent ensuite être utilisés pour paramétrer la déclaration de *l'entity*, ainsi que la déclaration de *l'architecture* :

```
entity RAM is
    generic(
        data_width : integer := 8; -- Taille des données, type entier,
                                   -- valeur par défaut 8 (optionnel)
        addr_width : integer := 5  -- Taille des addresses, type entier,
                                   -- valeur par défaut 5 (optionnel)
    );
    port(
        data_in  : in std_logic_vector(data_width-1 downto 0);
        addr     : in std_logic_vector(addr_width-1 downto 0);
        Wn_R     : in std_logic;
        CLK      : in std_logic;
        data_out : out std_logic_vector(data_width-1 downto 0)
    );
end RAM;
```

Lors de l'instantiation du composant, il est alors possible, de la même manière que l'instruction *port map*, d'appeler l'instruction *generic map* pour assigner de nouveaux paramètres :

```
RAM1 : entity work.RAM
generic map(
    data_width => 16,
    addr_width => 8
)
port map(
    data_in  => Di,
    addr     => A,
    Wn_R     => X,
    CLK      => CLK,
    data_out => Do
);
```

Travail demandé :

1. Ajoutez à votre projet un nouveau fichier shift_register_universal.vhd
2. Décrivez dans une *entity* nommée shift_register_universal le fonctionnement du registre prenant en *generic* la taille du registre
3. Ajoutez à votre projet deux nouveau fichiers de *testbench* tb_shift_register_universal4.vhd et tb_shift_register_universal16.vhd, dans lesquels vous décrirez les entrées à appliquer à deux registres universels à 4b et 16b dont vous ne changerez que le *generic map* (pour gagner du temps, vous pourrez reprendre votre testbench tb_shift_register_universal8.vhd)
4. Synthétisez et simulez vos registres à décalage

Annexe

Description d'une bascule D

```
Library ieee;
use ieee.std_logic_1164.all;

-- D flip flop (rising edge-triggered)
--
--  Symbol :             Characteristic table :
--
--  +---------+          +---++----+-----+
--  |         |          | D || Q+ | Qn+ |
-- -- D      Q--         +---++----+-----+
--  |         |          | 0 ||  0 |  1  |
-- --> CLK  Q*--         +---++----+-----+
--  |         |          | 1 ||  1 |  0  |
--  +---------+          +---++----+-----+
--

entity Dflipflop is
        port (
                -- Inputs
                D   : in std_logic; -- Data input
                CLK : in std_logic; -- Clock (rising-edge triggered)
                -- Outputs
                Q   : out std_logic; -- Flip flop output
                Qn  : out std_logic  -- Flip flop complemented output
                );
end Dflipflop;

architecture behavioral of Dflipflop is
begin
        process(CLK)
        begin
                if (CLK'event and CLK = '1') then
                        Q  <= D;
                        Qn <= not(D);
                end if;
        end process;
end behavioral;
```

Description d'une bascule D avec reset asynchrone

```
Library ieee;
use ieee.std_logic_1164.all;

-- D flip flop (rising edge-triggered)
-- With asynchronous reset (active low)
--
--  Symbol :             Truth table :
--
--  +---------+          +-----+------+---++----+-----+
--  |         |          | CLK | RSTn | D || Q+ | Qn+ |
-- -- D      Q--         +-----+------+---++----+-----+
--  |         |          |  ^  |   1  | 0 ||  0 |  1  |
-- --> CLK  Q*--         +-----+------+---++----+-----+
--  |         |          |  ^  |   1  | 1 ||  1 |  0  |
--  +----O----+          +-----+------+---++----+-----+
--       |               |  X  |   0  | X ||  X |  0  |
--      RSTn             +-----+------+---++----+-----+
--

entity Dflipflop is
        port (
                -- Inputs
                D    : in std_logic; -- Data input
                CLK  : in std_logic; -- Clock (rising-edge triggered)
                RSTn : in std_logic; -- Reset* (asynchronous)
                -- Outputs
                Q   : out std_logic; -- Flip flop output
                Qn  : out std_logic  -- Flip flop complemented output
                );
end Dflipflop;

architecture behavioral of Dflipflop is
begin
        process(CLK, RSTn)
        begin
                if (RSTn = '0') then
                        -- Asynchronous reset
                        Q  <= '0';
                        Qn <= '1';
                elsif (CLK'event and CLK = '1') then
                        -- D flip flop
                        Q  <= D;
                        Qn <= not(D);
                end if;
        end process;
end behavioral;
```