Synthèse de circuits logiques combinatoires Introduction au langage VHDL

Saphire 231: ELECTRONIQUE NUMERIQUE

10 Février 2025

Scott HAMILTON - Ibrahim ELKASSIMI

Table des matières

I Objectif

L'objectif de ce TP est de :

- Maîtriser l'environnement FPGA pour la programmation en VHDL.
- Concevoir des circuits logiques combinatoires .
- Simuler, synthétiser et valider le fonctionnement des circuits sur une carte FPGA.

II Mise en œuvre d'un additionneur 1 bit, 2 entrées (A et B), à propagation de retenue (C) : le full adder

II.1 Création d'un nouveau projet

À partir de la description donnée dans l'enoncé, on trouve les équation suivantes :

$$\begin{cases} S & = A \oplus B \oplus C_{in} \\ C_{out} & = A(B \oplus C_{in}) + BC_{in} \end{cases}$$

Ce que l'on peut schématiser avec un circuit logique.

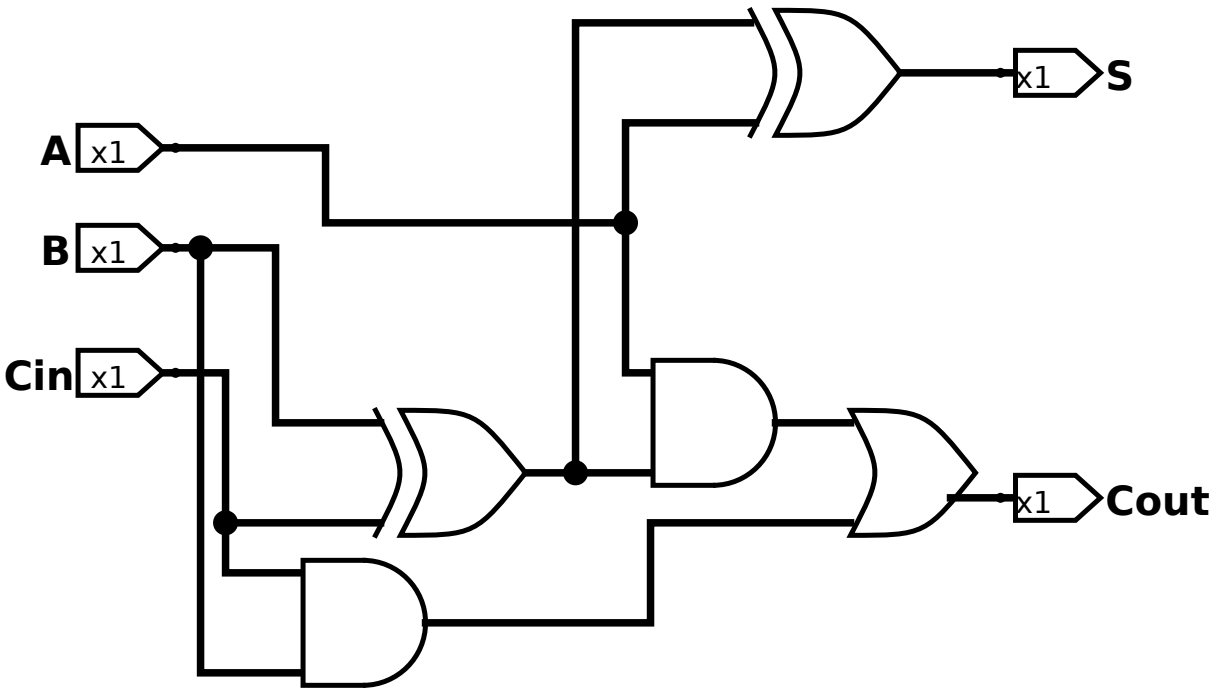

Figure 1 – Circuit logique d'un additionneur 1 bit.

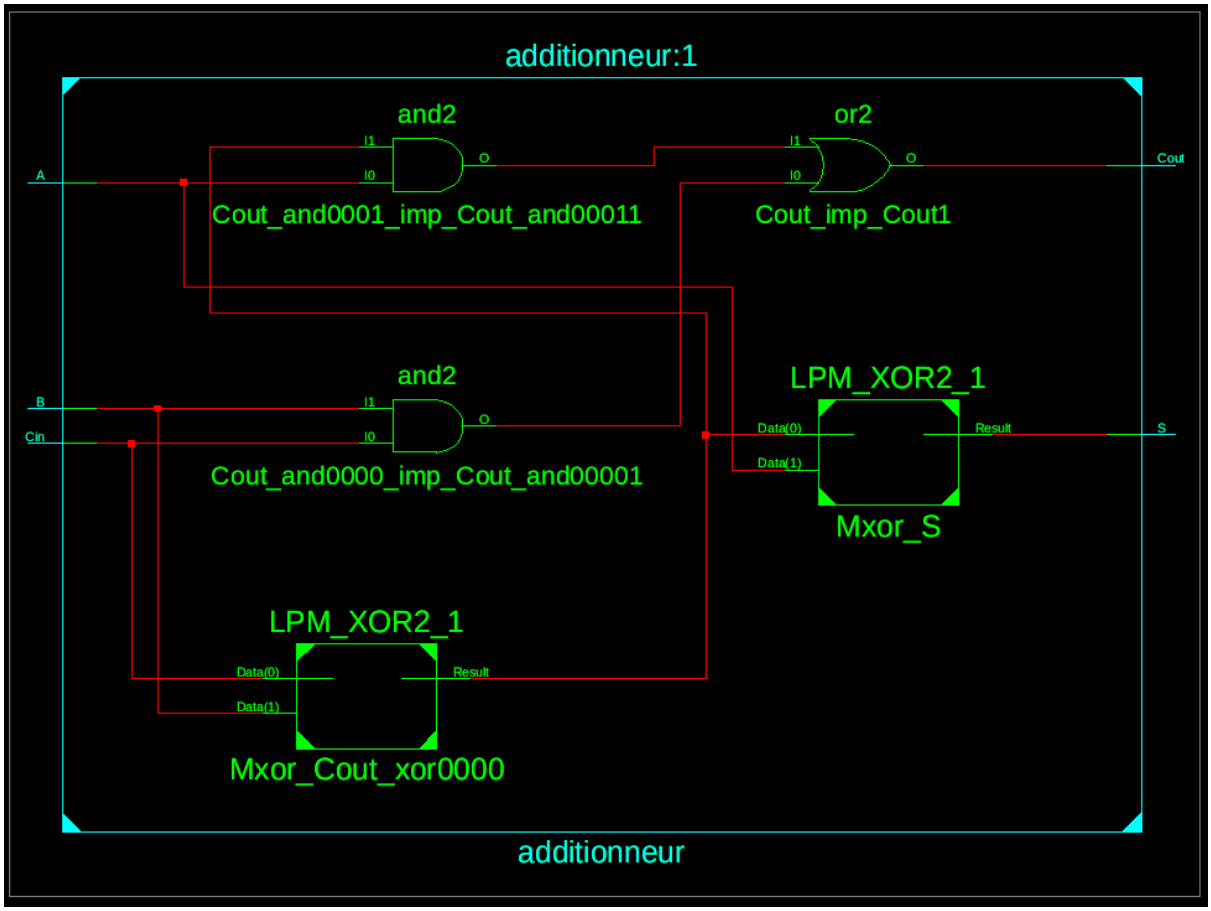

FIGURE 2 – Circuit synthétisé à partir du VHDL

On retrouve également sa table de vérité :

A	B	Cin	S (Sum)	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

FIGURE 3 – Table de vérité d'un additionneur 1 bit.

```
S <= (A xor B) xor Cin;
Cout <= (Cin and B) or (A and (B xor Cin));
```

II.2 Simulation

Une fois cette description saisie dans notre fichier, nous avons procédé à la simulation du code implémenté afin de vérifier le bon fonctionnement de l'additionneur. Pour ce faire, nous avons réalisé une simulation sur 1 microseconde en modifiant les états des 3 entrées toutes les 100 ns. Le code et le résultat de la simulation sont présentés ci-dessous :

```
88    -- Stimulus process
89    stim_proc: process
90    begin
91       -- hold reset state for 100 ns.
92       wait for 100 ns;
93       A  <='1';
94       B  <='0';
95       Cin<='0';
96       wait for 100 ns;
97       A  <='0';
98       B  <='1';
99       Cin<='0';
100      wait for 100 ns;
101      A  <='1';
102      B  <='1';
103      Cin<='0';
104      wait for 100 ns;
105      A  <='0';
106      B  <='0';
107      Cin<='1';
108      wait for horloge_period*10;
```

FIGURE 4 – Code de la simulation

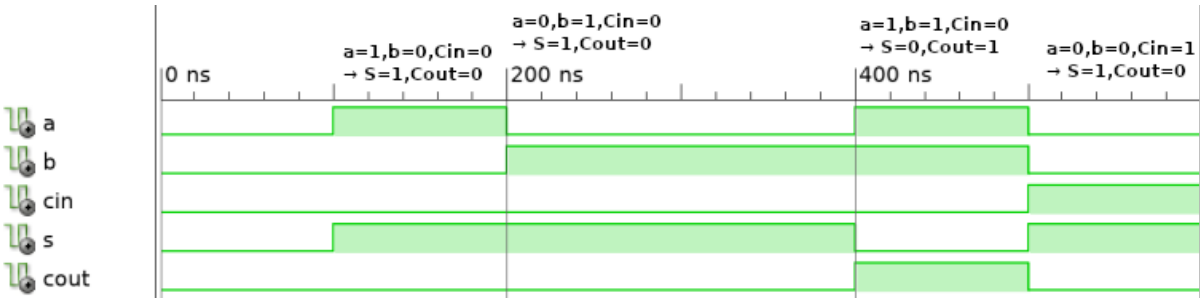

FIGURE 5 – Chronogramme de la simulation

II.3 Décodeur hexadécimal pour afficheur 7 segments.

Dans cette section, nous mettons en pratique les outils acquis lors de la première partie à travers l'exemple de l'additionneur. L'objectif est d'afficher un chiffre hexadécimal, codé sur 4 bits, sur un afficheur 7 segments à anodes communes.

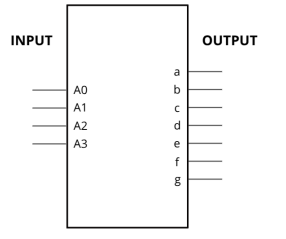

FIGURE 6 – L'entité du composant

II.3.1 Cas d'affectation simple :

Par contrainte de temps on a préféré passer au cas d'affectation sélective qui semble plus intéressant.

II.3.2 Cas d'affectation sélective :

En construisant la table de vérité ci-dessous, on démontre qu'il est possible de concevoir l'afficheur de manière rapide et efficace en utilisant une affectation conditionnelle ou sélective.

	A						C				
g	f	e	d	c	b	a	C(3)	C(2)	C(1)	C(0)	
1	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	1	0	0	0	1	1
0	1	0	0	1	0	0	0	0	1	0	2
0	1	1	0	0	0	0	0	0	1	1	3
0	0	1	1	0	0	1	0	1	0	0	4
0	0	1	0	0	1	0	0	1	0	1	5
0	0	0	0	0	1	1	0	1	1	0	6
1	1	1	1	0	0	0	0	1	1	1	7
0	0	0	0	0	0	0	1	0	0	0	8
0	0	1	1	0	0	0	1	0	0	1	9

FIGURE 7 – Table de vérité du décodeur hexadécimal pour afficheur 7 segments.

Nous avons également ajouté la possibilité de sélectionner un seul afficheur à l'aide de boutons poussoirs. La figure suivante illustre le bon fonctionnement de notre afficheur.

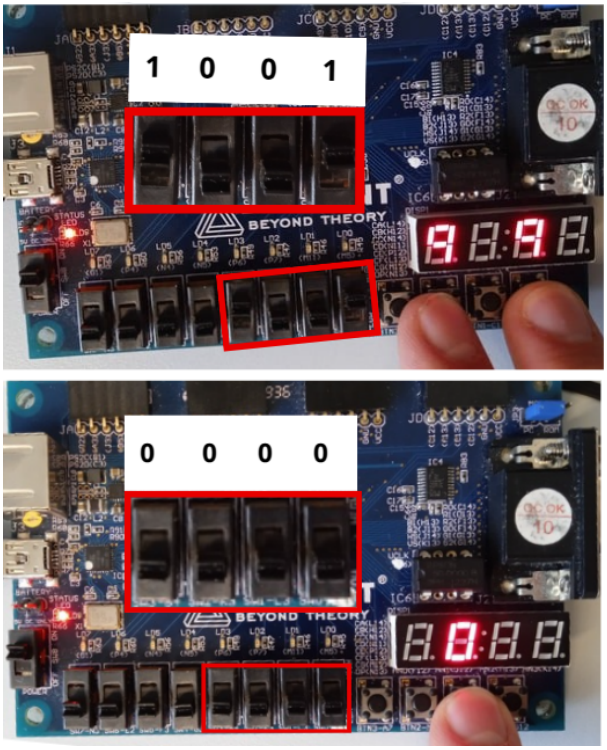

FIGURE 8 – Résultat du décodeur sur la carte.

III Codeur à priorité « 7 vers 3 »

Pour cette partie on suppose qu'on a sept capteurs C1, C2 ... et C7, ils sont classés par ordre de priorité croissante, du capteur 1 (priorité la plus faible) au capteur 7 (priorité la plus élevée). Ainsi, on peut établir la table de vérité suivante :

A0	A1	A2	C1	C2	C3	C4	C5	C6	C7
1	1	1	0	0	0	0	0	0	0
0	0	0	X	X	X	X	X	X	1
0	0	1	X	X	X	X	X	1	0
0	1	0	X	X	X	X	1	0	0
0	1	1	X	X	X	1	0	0	0
1	0	0	X	X	1	0	0	0	0
1	0	1	X	1	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0

FIGURE 9 – Table de vérité du codeur à priorité

Dans notre cas, chaque capteur est associé à un interrupteur. Le numéro du capteur prioritaire sera affiché sur l'afficheur 7 segments et indiqué (en binaire) également à l'aide des LEDs .

```
-- signal temporaire car on ne peut pas select
-- sur la sortie A pour construire G
signal Atmp : std_logic_vector(2 downto 0);

Atmp <=  "000" when C(0 downto 0) = "1"
 else "001" when C(1 downto 0) = "10"
 else "010" when C(2 downto 0) = "100"
 else "011" when C(3 downto 0) = "1000"
 else "100" when C(4 downto 0) = "10000"
 else "101" when C(5 downto 0) = "100000"
 else "110" when C(6 downto 0) = "1000000"
 else "111";
A <= Atmp;
WITH Atmp SELECT
  G <= "1000000" WHEN "000",
    "1111001" WHEN "001",
    "0100100" WHEN "010",
    "0110000" WHEN "011",
    "0011001" WHEN "100",
    "0010010" WHEN "101",
    "0000011" WHEN "110",
    "1111000" WHEN "111",
    "1111111" WHEN OTHERS;
```

Pour s'assurer du bon fonctionnement de notre code, nous testons différents scénarios. Deux d'entre eux sont présentés ci-dessous.

Premier test

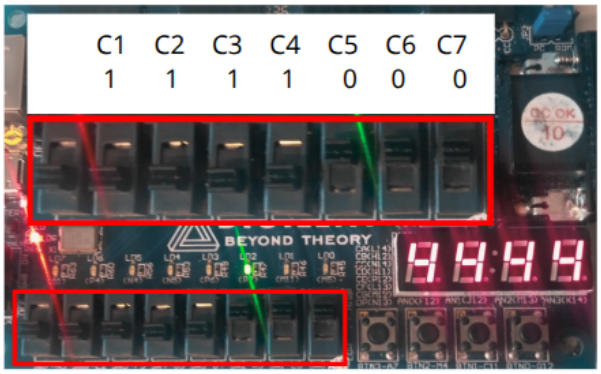

FIGURE 10 – Tests du codeur à priorité - 1

Les capteurs C1, C2, C3 et C4 demandent à transmettre une information. Étant le plus prioritaire, C4 est servi en premier. Le mot binaire généré est "100", ce qui allume la troisième LED en partant de la droite et affiche le numéro 4 sur tout les afficheurs 7 segments.

Second test

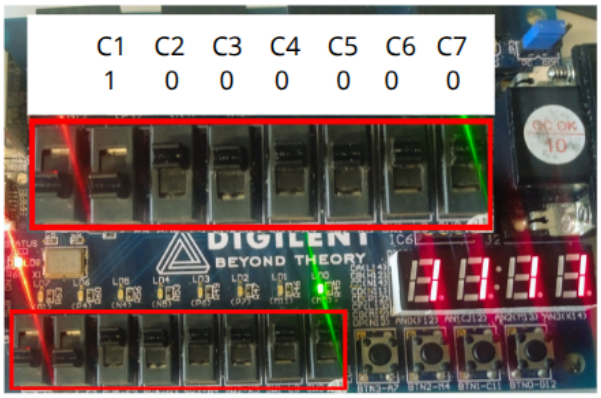

FIGURE 11 – Tests du codeur à priorité - 2

Seul le capteur C1 souhaite transmettre une information. Il est donc servi, et le mot binaire généré est "001". Ainsi, la première LED s'allume et le numéro 1 s'affiche sur l'afficheur 7 segments.