Rapport TP VHDL

Mise en œuvre d'une UART

I4SE-AE2

Avril 2005

Charles
Sébastien

SOMMAIRE

Introduction 3
I. Réalisation du module TX 3
A. Le protocole 3
B. Les Compteurs 4
1. Le compteur modulo 325 4
2. Le compteur modulo 16 5
3. Le compteur de bit 6
C. Le module Décalage 6
D. La partie Contrôle 7
E. Regroupement 9
F. Simulation 9
1. Le fichier TestBench 9
2. L'exécution générale 10
G. L'exécution réelle 10
II. Envoie d'un message préenregistré 12
A. La mémoire ROM 12
B. L'incrémenteur d'adresse 12
III. Mise en œuvre de l'IP UART 14
Conclusion 15

RAPPORT DE TP VHDL

MISE EN OEUVRE D'UNE UART

Introduction

L'objectif de ce TP est de réaliser une liaison série sur un FPGA. Egalement appelée UART (Universal Asynchronous Receiver/Transmitter), cette liaison série est capable d'envoyer une donnée parallèle bit par bit sur un bus. Elle est composée de deux modules principaux : le module émission TX et le module réception RX.

I. Réalisation du module TX

A. Le protocole

Le module TX permet de transmettre une donnée sur 8 bits sur la liaison série suivant un protocole imposé. Une trame série est composée d'un bit de start, un bit de stop, un bit de parité et des bits de données. Le bit de parité permet d'effectuer un contrôle sur la donnée reçue. Dans notre cas, nous ne l'utiliserons pas. Lorsque aucune donnée n'est envoyée, le bus reste à l'état haut.

START	D0	D1	D2	D3	D4	D5	D6	D7	STOP

Trame série

La vitesse de transmission est de 9600 bauds (bits par secondes) pour une horloge du FPGA de 50 Mhz.
Chaque bit prend donc 104 µs pour être transmit. La fréquence de l'horloge étant de 50 Mhz, la période de l'horloge est de 20 ns, ce qui implique qu'il faut 5200 tops d'horloge pour transmettre un bit. On utilisera pour cela deux compteurs : un compteur modulo 16 et un compteur modulo 325. On obtient ainsi un compteur global de 16*325 = 5200. Une machine d'état contrôle le transfert en gérant les différents modules, dont le module de décalage qui permet de transformer l'information parallèle en information série.

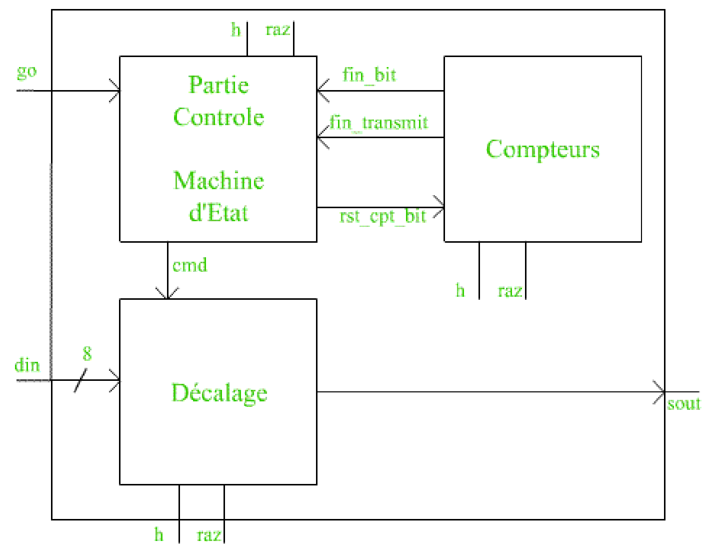


Schéma global du FPGA

Les trois entités présentées seront ensuite introduites dans un fichier global qui contiendra les entités ainsi que les branchements entre les entités

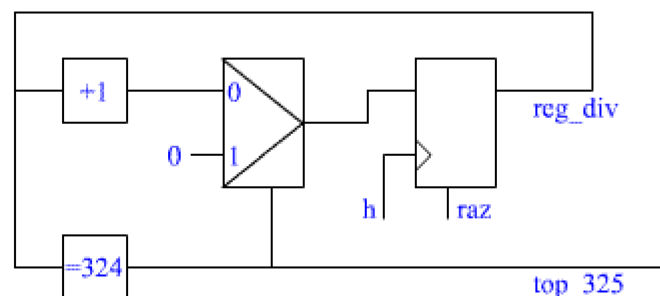
B. Les Compteurs

Le module Compteurs est composé de 3 parties :

- 1 compteur modulo 16
- 1 compteur modulo 325
- 1 compteur modulo 10

1. Le compteur modulo 325

Voici le schéma RTL du compteur :



Compteur module 325

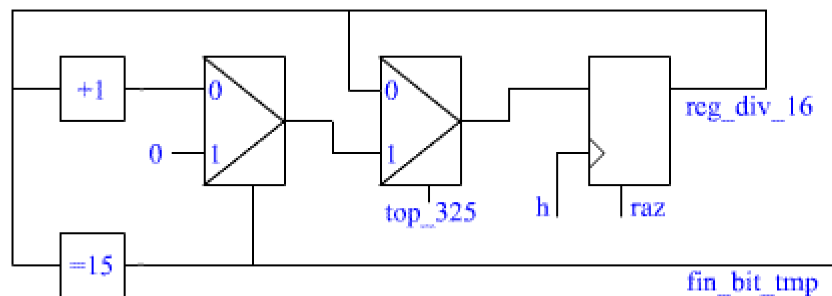
Reg_div est une variable du compteur de type ‘natural’ qui s’incrémente à chaque front montant de l’horloge. Arrivé à 324, le compteur envoie un signal *top_325* au compteur modulo 16 pour l’incrémenter et se remet à zéro grâce à un multiplexeur.

Code :

```
-- Diviseur d'horloge (/325)
process(h,raz) is
variable reg_div : natural range 0 to 324;
begin
	if raz = '1' then reg_div := 0;
	elsif rising_edge(h) then
		if reg_div = 324 then reg_div := 0; top_325 <= '1';
		else reg_div := reg_div + 1; top_325 <= '0';
		end if;
	end if;
end process;
```

2. Le compteur modulo 16

Très semblable au compteur précédent, ce compteur s'incrémente lorsque le signal top_325 est à 1. Arrivé à 15, le processus envoie le signal *fin_bit_tmp* au compteur modulo 10 ainsi qu'à la machine d'état du système. Comme cette dernière ne fait pas partie de la même entité, il faut utiliser un signal interne *fin_bit_tmp* différent de la sortie *fin_bit*.



Compteur modulo 16

Pour brancher le signal sur la sortie, on utilise une instruction concurrente du process :

```
	fin_bit<=fin_bit_tmp;
```

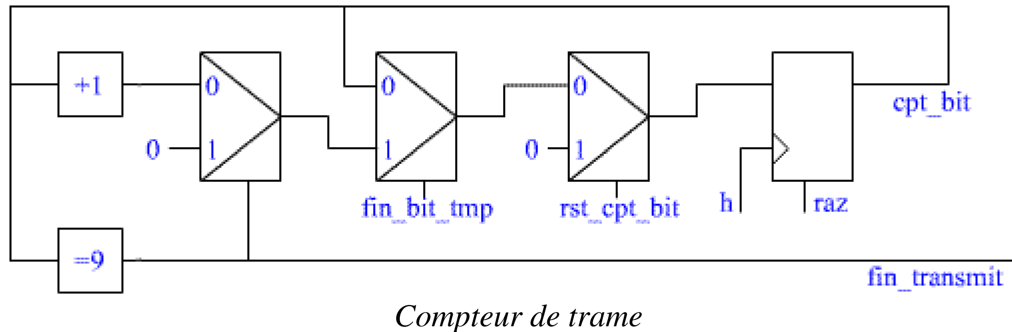
Code :

```
-- Diviseur de diviseur d'horloge (/16)
process(h,raz) is
variable reg_div_16 : natural range 0 to 15;
begin
	if raz = '1' then reg_div_16 := 0;
	elsif rising_edge(h) then
		if top_325 = '1' then
			if reg_div_16 = 15 then reg_div_16 := 0; fin_bit_tmp <=
'1';
			else  reg_div_16 := reg_div_16 + 1; fin_bit_tmp <= '0';
			end if;
		else reg_div_16 := reg_div_16; fin_bit_tmp <= '0';
		end if;
	end if;
end process;

fin_bit<=fin_bit_tmp;
```

3. Le compteur de bit

Contrairement aux compteurs précédents, ce compteur possède une remise à zéro synchrone contrôlée par la machine d'état du système via le signal *rst_cpt_bit*. Lorsque *rst_cpt_bit* est à l'état bas, le compteur s'incrémente lorsque *fin_bit_tmp* est égal à '1'.



Compteur de trame

Code :

```
-- Compte le nombre de bit's
process(h,raz) is
variable cpt_bit : natural range 0 to 10;
begin
	if raz = '1' then cpt_bit := 0;
	elsif rising_edge(h) then
		if rst_cpt_bit = '1' then
			cpt_bit := 0;
		elsif fin_bit_tmp = '1' then
			if cpt_bit = 9 then cpt_bit := 0; fin_transmit <= '1';
			else cpt_bit := cpt_bit + 1; fin_transmit <= '0';
			end if;
		else cpt_bit := cpt_bit; fin_transmit <= '0';
		end if;
	end if;
end process;
```

C. Le module Décalage

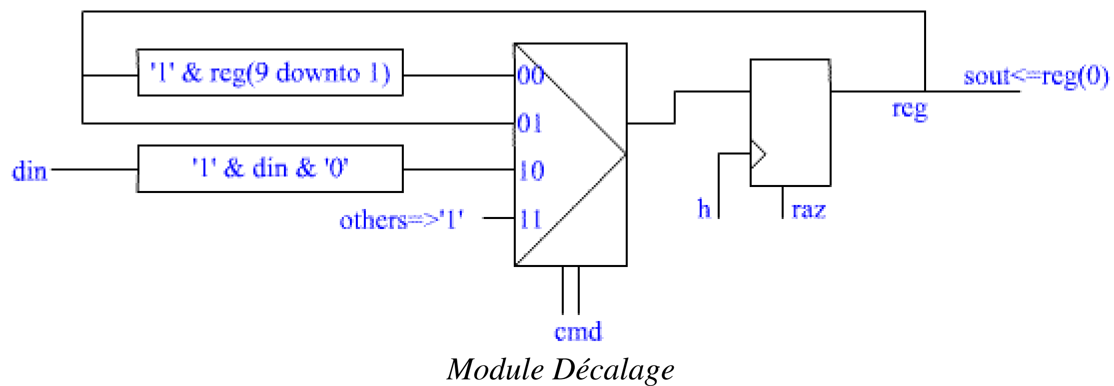
L'entité *Décalage* permet d'effectuer 4 actions sur la donnée en fonction du signal de commande *cmd* envoyée par la partie contrôle du système.

Cmd	Action
00	Décalage du registre de sortie
01	Mémorisation de la valeur sur la sortie
10	Chargement de la donnée entrante
11	Remise à un du registre

Tableau des commandes

La donnée parallèle a envoyer se trouve sur le bus *din* composé de 8 bits. Pour pouvoir émettre une première valeur, il faut en premier lieu charger la valeur de *din* en rajoutant les bits de Start et de Stop propres à la liaison série dans un registre que nous appellerons *reg*. La valeur de sortie sera le bit 0 du registre. Comme la valeur doit rester sur le fil pendant un

temps dépendant de la vitesse de communication, une mémorisation est effectuée lorsque la commande *cmd* est à ‘‘01’’.



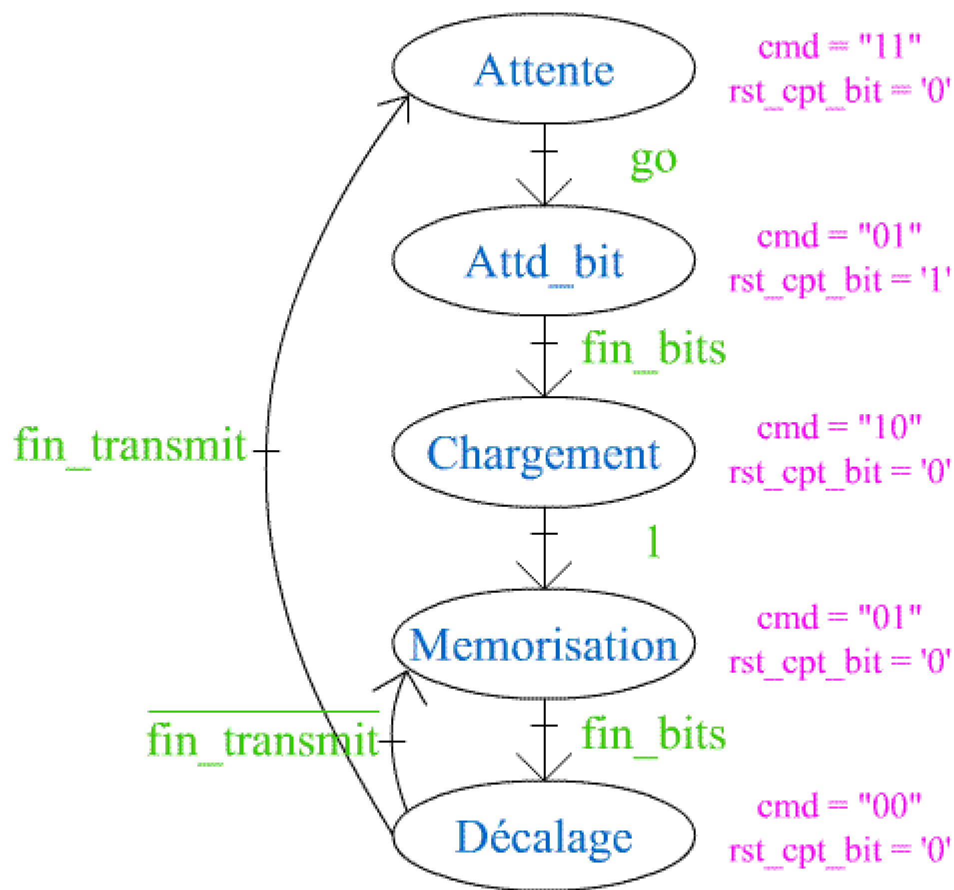
Module Décalage

Code du process:

```
--Decalage
process (h,raz) is
begin
      if raz = '1' then reg <= (others => '1');
      elsif rising_edge(h) then
            case cmd is
                  when "00" => reg <= '1'& reg(9 downto 1); -- Decalage
                  when "10" => reg <= '1'& din & '0'; -- Formatage -> Bits
de Start et de Stop
                  when "01" => reg <= reg; -- Ne fait rien. -> Continu la
transmission
                  when "11" => reg <= (others => '1');
                  when others => reg <= (others => '1');
            end case;
      end if;
end process;
```

D. La partie Contrôle

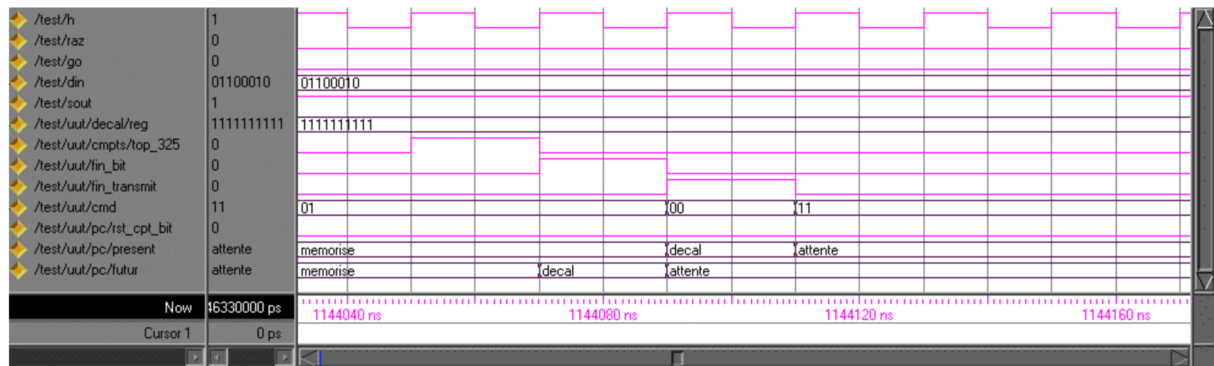
La partie contrôle s’effectue grâce à une machine d’états. Voici son diagramme :



Machine d'états

- Attente : Durant cet état, la sortie sout est à l'état haut. Le module attend le signal go pour passer à l'état suivant
- Attd_bit : Cet état effectue un reset du compteur de bit modulo 10. Le module attend ensuite que le compteur modulo 16 soit revenu à zéro (indiqué par le signal fin_bit) pour passer à l'état suivant et pour libérer le compteur.
- Chargement : Une fois le système prêt, l'entité *décalage* charge la valeur de la donnée parallèle *din* dans le registre *reg* grâce à la commande cmd ''10''. Une fois la valeur chargée, la machine d'état rentre dans une boucle jusqu'à ce que les 10 bits de l'information soient transmis.
- Mémorisation : L'état attend que la valeur mise sur sout soit finie de transmettre. Ceci est indiqué par le signal fin_bit.
- Décalage : Le décalage de la donnée parallèle est effectué grâce à la commande ''00''. Si tous les bits sont transmis, la machine d'état retourne dans l'état *Attente* sinon, elle recommence la boucle Mémorisation/Décalage.

A la fin de l'exécution, la machine d'état fait un dernier décalage avant de retourner dans l'état attente. Cependant, ce décalage ne génère aucune erreur sur le signal de sortie car à ce niveau tous les bits de *reg* sont à 1.
On observe sur le chronogramme de simulation suivant le passage de l'état *decal* à l'état *attente.*



Chronogramme de simulation

E. Regroupement

Pour regrouper les différentes entités ainsi créées, il suffit d'ajouter un nouveau fichier VHDL dans le projet et de les déclarer à l'intérieur de l'architecture selon la syntaxe.
Par exemple, pour insérer la machine d'état :

```
PC : entity work.StateMachine
  port map(
            --IN
            h => h,  -- horloge
            raz => raz,  -- RAZ actif haut
      go => go, -- Demande Debut de transmission
      fin_bit => fin_bit, -- Indique la fin d'un bit
      fin_transmit => fin_transmit, -- Indique la fin de la transmission
totale

            --OUT
      cmd => cmd, -- Selectionne le mode de fonctionnement
      rst_cpt_bit => rst_cpt_bit); -- remet le compteur de bits a ZERO);
```

Pour faire la liaison entre les entités, il est nécessaire de créer des signaux intermédiaires.

```
signal fin_bit,fin_transmit,rst_cpt_bit : std_logic;
signal cmd : std_logic_vector(1 downto 0);
```

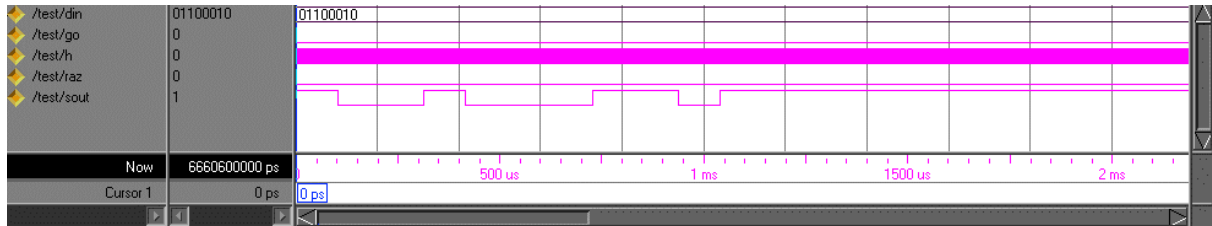
F. Simulation

1. Le fichier TestBench

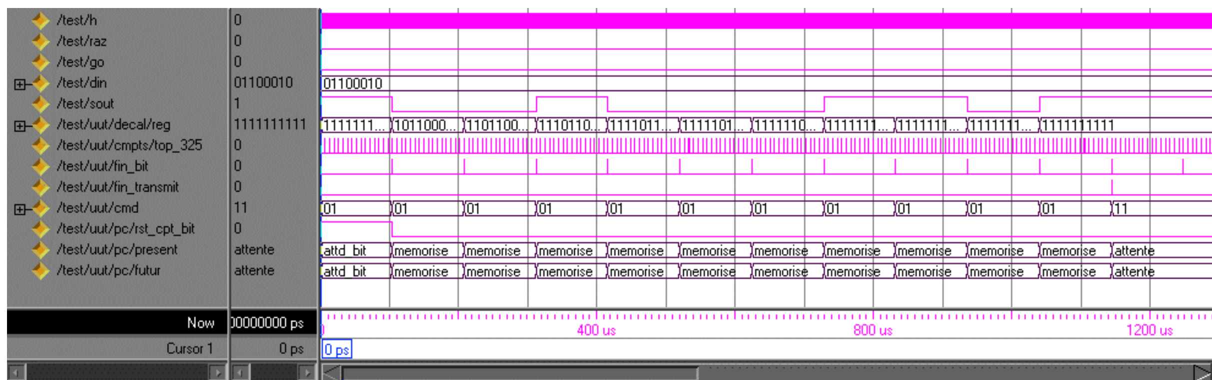
Pour effectuer une simulation du code VHDL, il faut écrire un fichier de type test bench qui permet de simuler une activité extérieure au module de transmission.
Le test bench est en fait une entité dans laquelle est inclus le module TX. On génère ensuite les signaux *raz* et *h*, puis les entrées *din* et *go*.

2. L'exécution générale

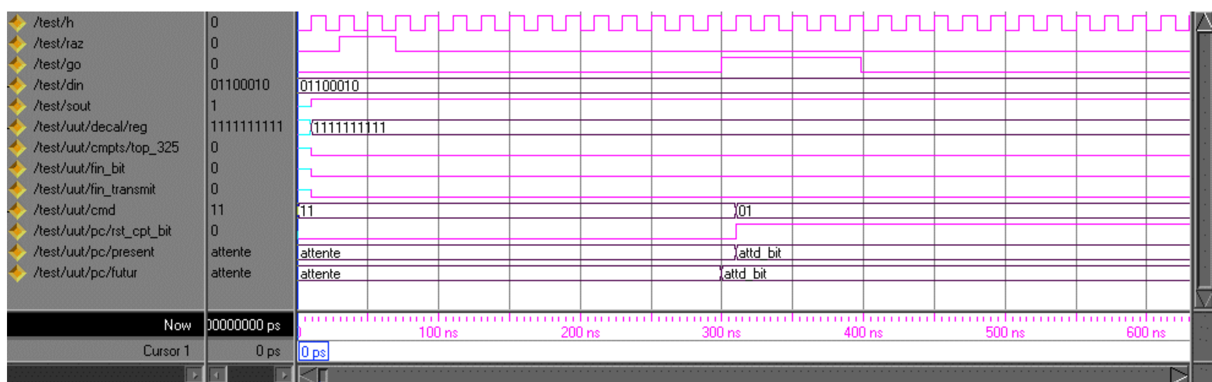
L'exécution de la simulation donne ce chronogramme. Avec une valeur pour *din* de 01100010 et après avoir fait un créneau avec le signal *go*, on obtient sur *sout* les valeurs attendues.



Le module semble donc être correcte. En observant l'ensemble des signaux du système, on peut contrôler le déroulement de la machine d'état ainsi celui des compteur. Il y a bien 10 mémorisations de la valeur, ce qui signifie qu'il y a bien eu 10 décalages. Une fois terminé, le système retourne bien dans l'état *attente*.



En zoomant sur le démarrage, on s'aperçoit que tous les signaux sont définit dès le premier front montant de l'horloge. Le système démarre bien dans l'état initial voulu. Le raz n'a pas d'action visible, mais il remet notamment à zéro tous les compteurs.



G. L'exécution réelle

Avant d'effectuer la programmation, il faut établir un fichier de contraintes qui permet de relier chaque signal d'entrée/sortie avec une broche du FPGA. Ainsi, la donnée *din* sera reliée

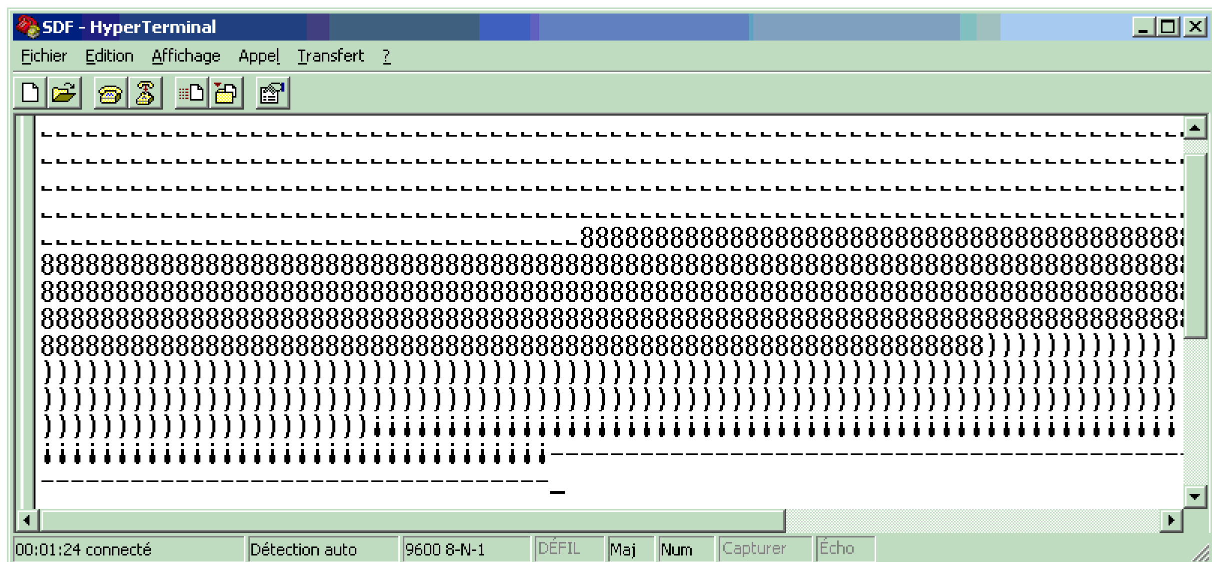
aux switchs de la carte d'expérimentation et deux autres boutons serviront à faire le *raz* ainsi que le *go*.

Design Object List - I/O Pins

	I/O Name	I/O Direction	Loc	Bank	I/O Std.
	din<0>	Input	K13	BANK	
	din<1>	Input	K14	BANK	
	din<2>	Input	J13	BANK	
	din<3>	Input	J14	BANK	
	din<4>	Input	H13	BANK	
	din<5>	Input	H14	BANK	
	din<6>	Input	G12	BANK	
	din<7>	Input	F12	BANK	
	go	Input	L14	BANK	
	h	Input	T9	BANK	
	raz	Input	L13	BANK	
	sout	Output	R13	BANK	

Définition des pattes du composant

L'exécution observée correspond bien aux attentes. En ouvrant le logiciel 'HyperTerminal', on peut observer les données reçus par le port série de l'ordinateur. A chaque fois que le bouton *go* est appuyé, le logiciel reçoit un ensemble de caractères. Cet ensemble correspond au nombre de fois que la donnée est envoyée par le FPGA. Comme nous n'effectuons pas de détection de front, la boucle de la machine d'état a le temps de s'exécuter plusieurs fois pendant que le bouton reste appuyé.



Réception série sous HyperTerminal

II. Envoie d'un message préenregistré

L'objectif de cet exercice est de mettre en œuvre le programme développé précédemment pour qu'il envoie un message préenregistré dans une mémoire ROM. L'envoie du message est déclenché par le bouton *go*.

A. La mémoire ROM

Pour créer une zone mémoire dans le FPGA, il est possible de créer un tableau de valeurs. Voici un exemple de mémoire composé de 16 valeurs de 8 bits :

```
type tableau is array (0 to 15) of std_logic_vector(7 downto 0);
```

Les valeurs sont ensuite définit dans un tableau de constante.

```
constant rom:tableau :=
(X"48",X"41",X"50",X"50",X"59",X"20",X"4E",X"45",X"57",X"20",X"59",X"45",X"
41",X"52",X"20",X"20");
```

rom possède l'adresse début du tableau et rom(integer var) pointe vers l'index var du tableau.

Ainsi, lorsque l'on fait :

```
din <= rom(to_integer(unsigned(adr)));
```

La valeur du tableau à l'index *adr* est mise sur le bus *din*.

B. L'incrémenteur d'adresse

Pour effectuer l'envoie du message, il faut incrémenter la valeur de l'index du tableau après chaque envoi. Il faut donc insérer un compteur qui s'incrémente à chaque fois que le signal *fin_transmit* est à l'état haut.

Code du compteur :

```
--Incremente l'adresse
process (h,raz) is
begin
      if raz = '1' then adr <= (others => '0');
      elsif rising_edge(h) then
            if fin_transmit = '1' then adr <=
std_logic_vector(unsigned(adr)+1);
            else adr <= adr;
            end if;
      end if;
end process;
```

Le signal *adr* contient l'index du tableau. Comme ce dernier ne contient que 16 valeurs, *adr* n'a besoin que de 4 bits. Le retour à zéro se fait automatiquement car *adr* est un std_logic_vector.

On transforme ensuite la valeur du tableau pour la mettre sur le bus *din*. Contrairement à l'architecture précédente, *din* n'est plus un port d'entrée, mais un signal interne qui entre directement dans l'entité de décalage.

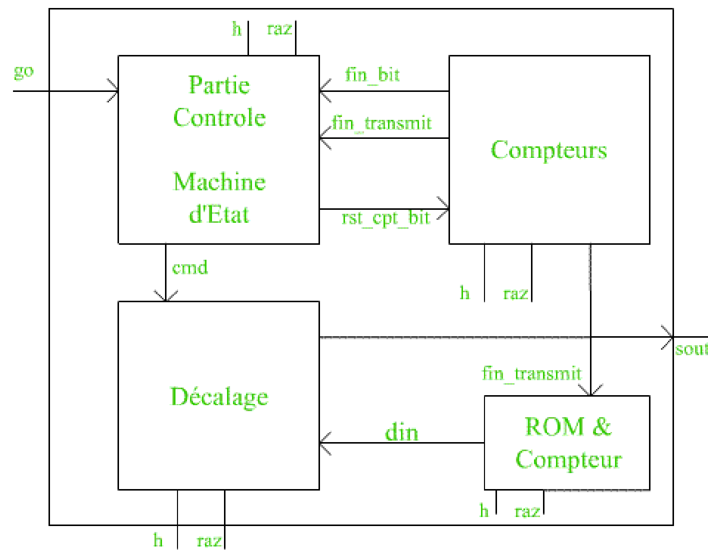
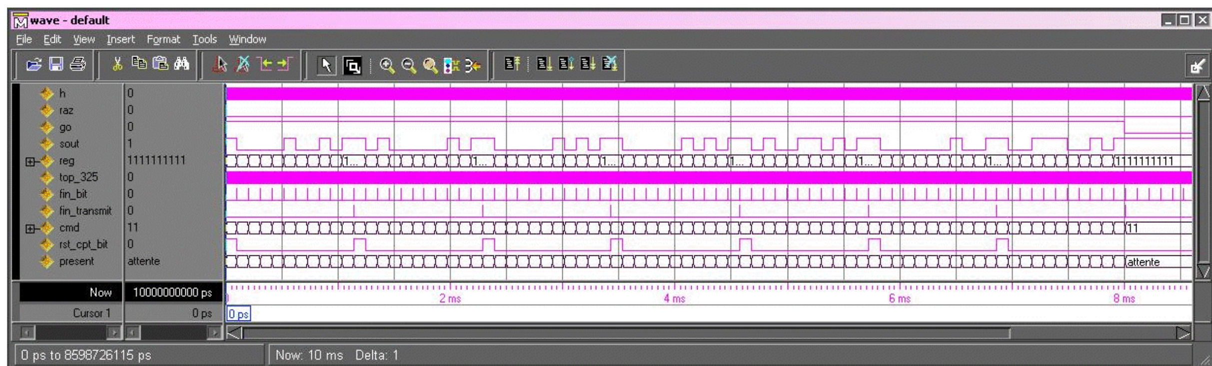


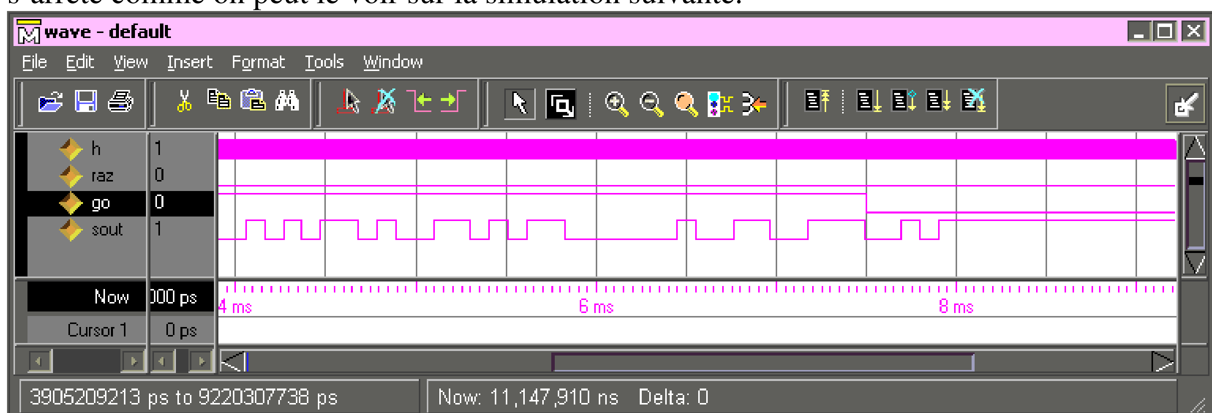
Schéma global de l'architecture

En simulation, on observe le comportement théorique de l'architecture, c'est-à-dire l'envoi en continu de valeurs sur *sout*, tant que *go* est à l'état haut.



Simulation

Lorsque *go* retombe à l'état bas, l'octet qui était en train d'être transmis se termine et l'envoi s'arrête comme on peut le voir sur la simulation suivante.



Simulation

III.Mise en œuvre de l'IP UART

L'objectif de cet exercice est d'utiliser une application déjà faite : une IP (Intellectual Property). L'IP proposée réalise les modules RX et TX d'une liaison série. Nous allons donc l'utiliser pour effectuer un affichage de la valeur des switchs de la carte d'expérimentation sur l'ordinateur en utilisant la partie TX ainsi qu d'afficher sur les leds, la valeur que l'on envoie sur la carte à partir du logiciel HyperTerminal du PC.

L'intérêt d'utiliser une IP est multiple. Premièrement, elle permet de gagner du temps car il existe des IPs très complexes, payantes ou gratuites et deuxièmement, elles garantissent une certaines fiabilité ainsi qu'une très bonne portabilité par rapport à un code développé personnellement. Cependant, il est plus difficile d'optimiser une IP car certains modules internes peuvent être inutile pour l'application à développer.

L'IP UART possède un ensemble de paramètres génériques qui permettent de définir la vitesse de transmission ainsi que la présence de bits de stop, de parité ou autres.

Voici le schéma de l'architecture réalisée.

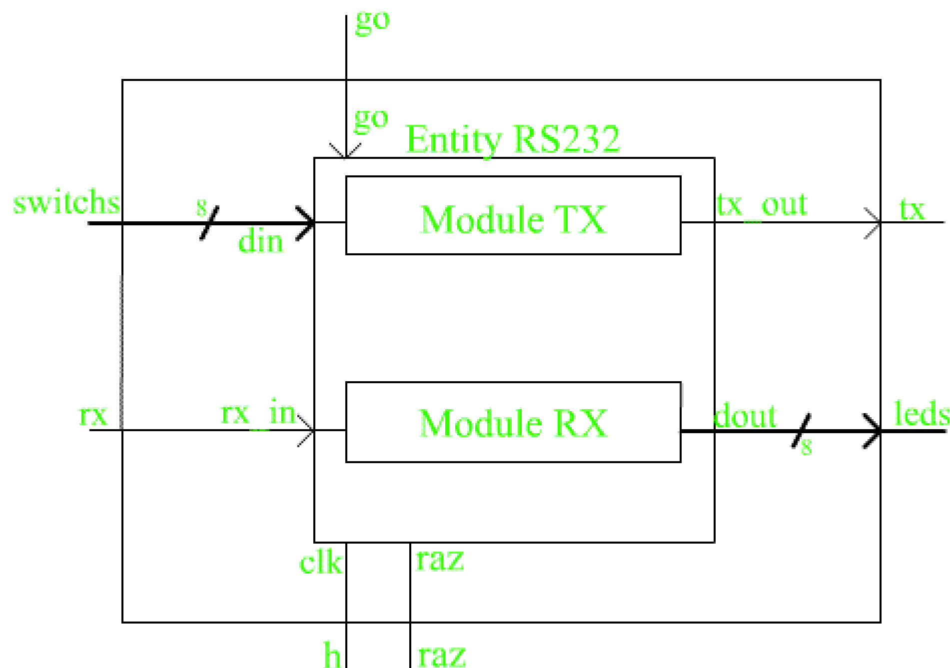
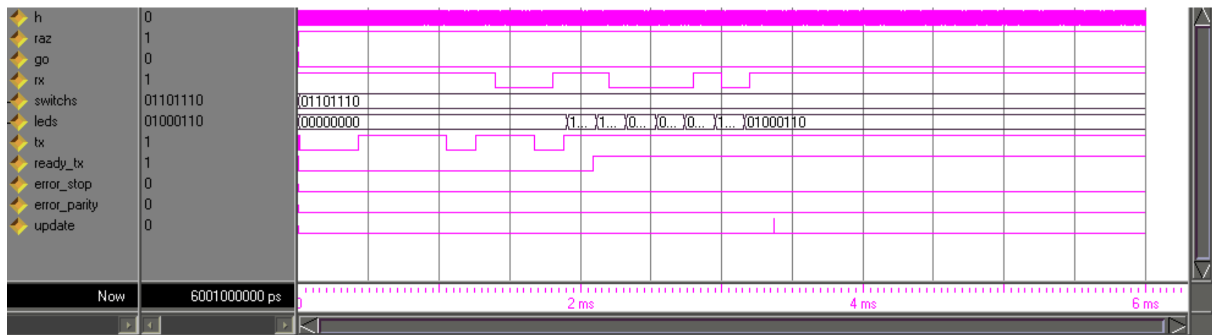


Schéma global de l'architecture avec IP

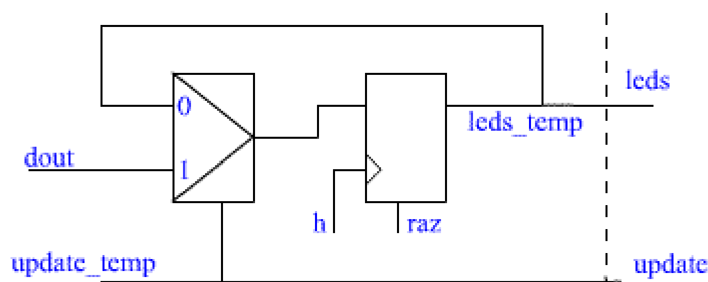
Pour cela il suffit d'insérer l'entité de l'IP dans l'architecture de notre projet puis de brancher les entrées et les sorties aux ports correspondants.

Lorsque l'on simule l'architecture avec une réception et une émission, l'émission s'effectue correctement mais, par contre, pendant la réception, la sortie *leds* n'est pas stable. La fin de la réception est indiquée par un créneau du signal *update*. Pour avoir une sortie stable il faut donc rajouter un état séquentiel qui mémorise la valeur lorsque *update* passe à 1.



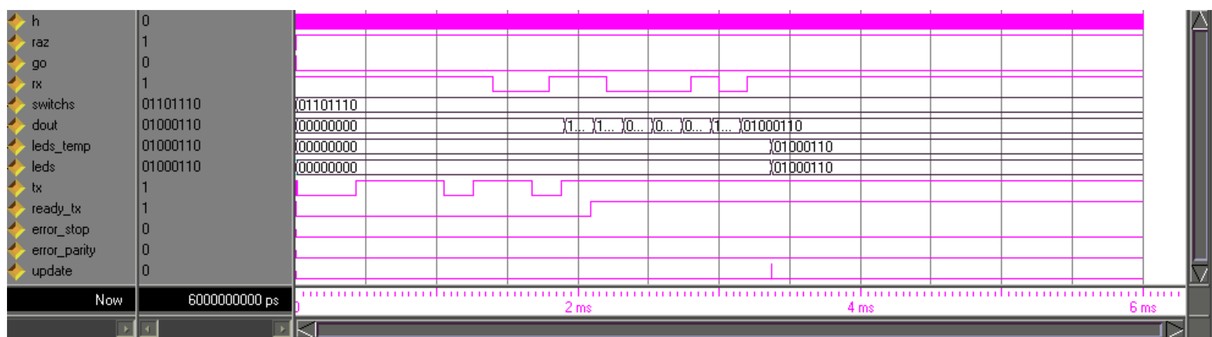
Simulation avec leds instables

Cependant, la vitesse de changement des valeurs est tellement rapide que, pour des leds, l'utilisateur ne voit pas la phase d'instabilité.



Etage de mémorisation de dout -> leds

Sur le chronogramme suivant, on peut voir que la sortie *leds* prend sa valeur quand *update* passe à l'état haut.



Simulation avec leds stables

Conclusion

Ce TP nous a permis de mettre en œuvre les connaissances acquises en VHDL depuis la deuxième année de l'ESIEE. Nous avons réalisé un module de transmission UART de manière peu modulaire, mais correspondant au cahier des charges imposé. L'utilisation d'une IP nous a appris à utiliser un code que nous n'avons pas programmé et à l'implanter dans une architecture en se referant à la documentation et à l'observation de l'exécution.

ANNEXES

Code de :

Entité Principale du module TX avec switchs
Entité Compteurs
Entité Décalage
Entité Machine d'états
Entité Principale du module TX avec ROM
Entité Principale du module avec IP UART
Test Bench de l'entité principale du module avec IP UART