# Compte Rendu : Émetteur pour Liaison Numérique

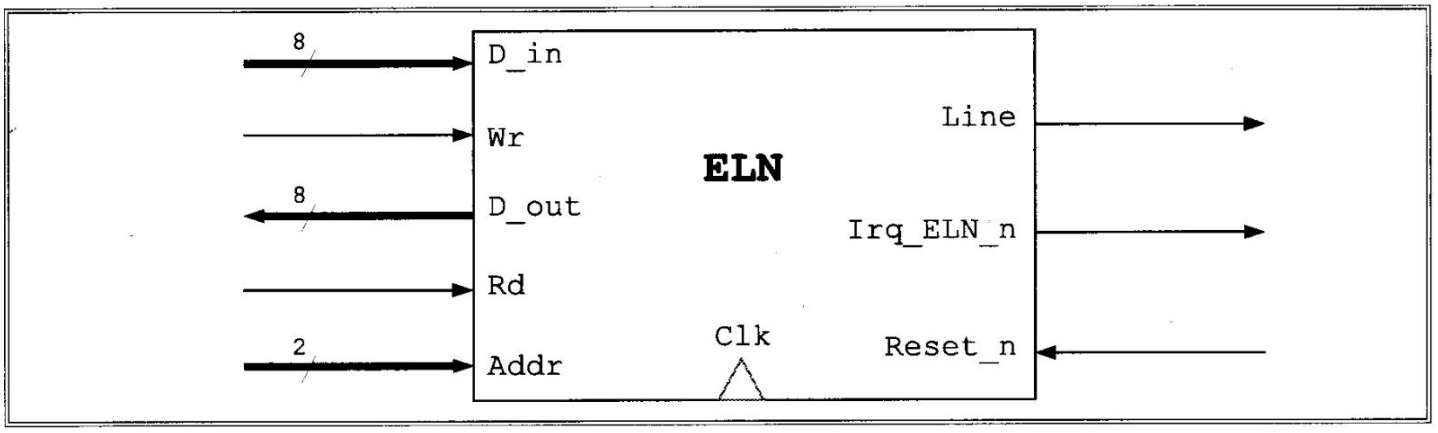
## I) Spécificité du composant

Le composant à concevoir est un émetteur pour l'envoi de données sous forme de trames sur une liaison numérique. Ce composant devra être intégrable dans un module système incluant un processeur NiosII et une interface d'interconnexion Avalon.

### a) Vue externe du composant

Les ports nécessaires sont, en plus de l'interface Avalon (***D_in***, ***D_out***, ***Wr***, ***Rd***, ***Addr***), une sortie d'interruption active à l'état bas (***Irq_ELN_n***), une entrée de réinitialisation active à l'état bas (***Reset_n***) et le port de sortie qui transmettra les trames sérialisées et encodées (***Line***).

Le composant peut donc être représenté tel que suit :




### b) Registres du périphérique

L'interface d'interconnexion Avalon fonctionne avec plusieurs registres en lecture et/ou en écriture. Ces registres permettent d'écrire les données à sérialiser (***ELNFIFO***), à lire l'état du composant (***ELNStatus***), ou à le contrôler (***ELNControl***).

Ces 3 registres sont organisés de la manière suivante :

| Registre | Offset | Accès | Format MSB … LSB |
|---|---|---|---|
| ELNStatus | 0 | R | - \| - \| - \| - \| Irq \| Full \| Empty \| Busy |
| ELNControl | 1 | R/W | - \| - \| - \| - \| - \| - \| IrqEn \| Start |
| ELNFifo | 2 | W | Data |

Cependant, comme les bits émis sur la ligne ***Line*** sont cadencés à un rythme fixe égal à $f_{clk}$/*clock_div*, je décide d'ajouter sur le registre de contrôle 6 bits qui permettront de modifier la valeur de *clk_div*. Les registres deviennent donc :

| Registre | Offset | Accès | Format MSB … LSB | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| ELNStatus | 0 | R | - | - | - | - | Irq | Full | Empty | Busy |
| ELNControl | 1 | R/W | Clock_Div | | | | | | IrqEn | Start |
| ELNFifo | 2 | W | Data | | | | | | | |

## c) Fonctionnement interne du composant

Maintenant que la manière dont le composant interagit vers l'extérieur est connue, il faut définir son fonctionnement interne. Dans l'objectif, les données écrites sur ***ELNFIFO*** seront stockées dans une pile FIFO en attendant d'être sérialisées. Pendant ce temps, des fanions de start seront émis en boucle.

À la mise à '1' du bit de start, l'unité de contrôle commencera en même temps le calcul du CRC (Cyclic Redundancy Check) et la sérialisation/codage des données.

Comme le signal ***Line*** est cadencé au plus deux fois moins vite que l'horloge du système et donc que la vitesse de calcul du CRC, ce dernier sera toujours achevé quand l'unité de contrôle commandera sa sérialisation.

Sur ***Line***, les données (FIFO + CRC) sont codées par changement d'état avec des bits de stuffing. Quand le niveau '0' est à écrire, le niveau de ***Line*** change d'état. Si c'est un niveau '1', le niveau ne change pas.

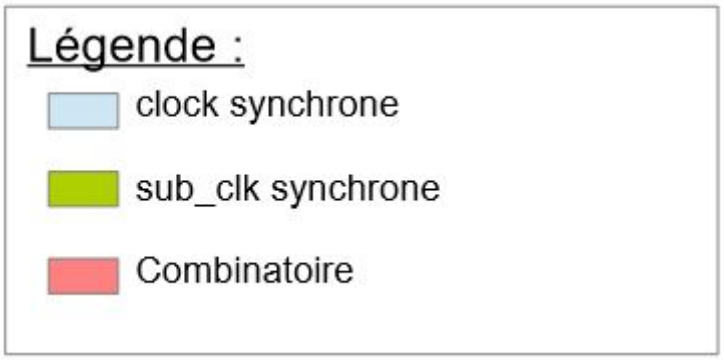
De plus, si 5 '1' consécutifs sont détectés sur un octet à transmettre (hors fanion de start), alors un '0' est inséré avant le cinquième '1' afin de ne pas nuire à la synchronisation du récepteur qui utilise les fronts du signal reçu.

Une fois la pile FIFO vide, le CRC est sérialisé puis l'émission de fanions de start recommence.

Je peux donc déjà utiliser une pile FIFO, une unité de contrôle (UC) gérée avec une Machine à États, une entité de sérialisation, et un diviseur d'horloge. Je peux également penser à des registres pour ***ELNControl*** dû à son accès en lecture et en écriture.

# II) Schémas

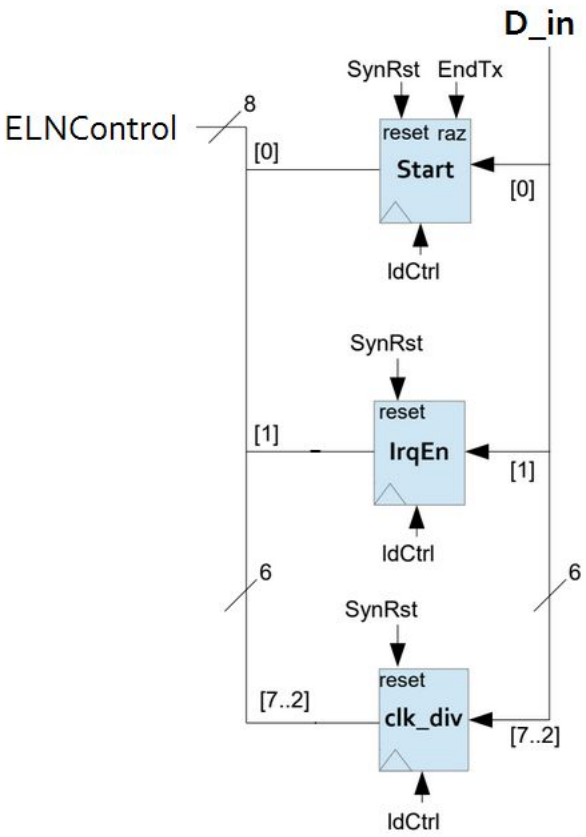
Dans la représentation des schémas suivants, j'adopte le code couleur :




Les schémas complets en format ".pdf" A3 sont joints en annexe I.

## a) Registre du control

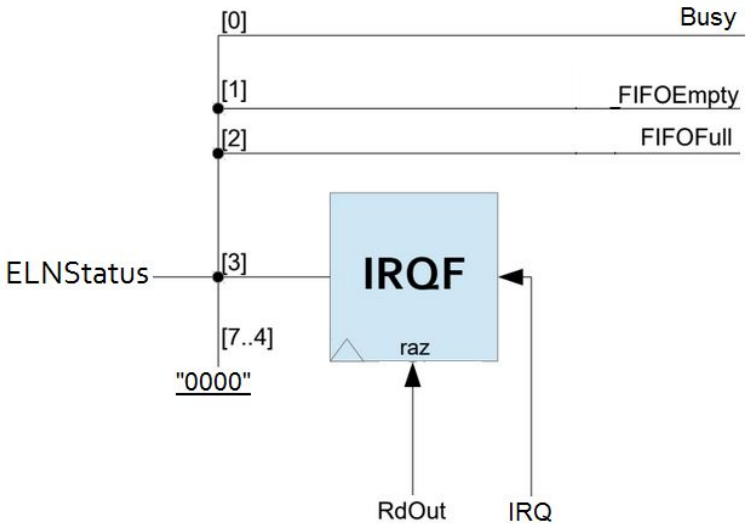
***ELNControl*** est en accès *R/W*, donc comme dit précédemment, je peux utiliser des registres pour chaque donnée qu'il comporte.
Je crée donc un registre 1-bit pour *Start*, un autre pour *IrqEn*, un registre 6-bit pour *clk_div*.
Ces registres prendront en entrée les bits concernés sur le bus ***D_in*** et recomposeront en sortie le registre ***ELNControl***, prêt à être lu.




Ces registres chargeront leur valeurs quand *ldCtrl*, signal créé par le décodeur de l'interface Avalon (cf. *d) Interface Avalon*) sera actif. Tous les registres seront réinitialisés au signal *SynRst*, généré à partir de ***Reset_n***, respectivement à 0, 0, et "001000" (8). *Start* est également remis automatiquement à zéro à la fin d'une transmission, signalé par le bit *EndTx*.

## b) Registre de Status

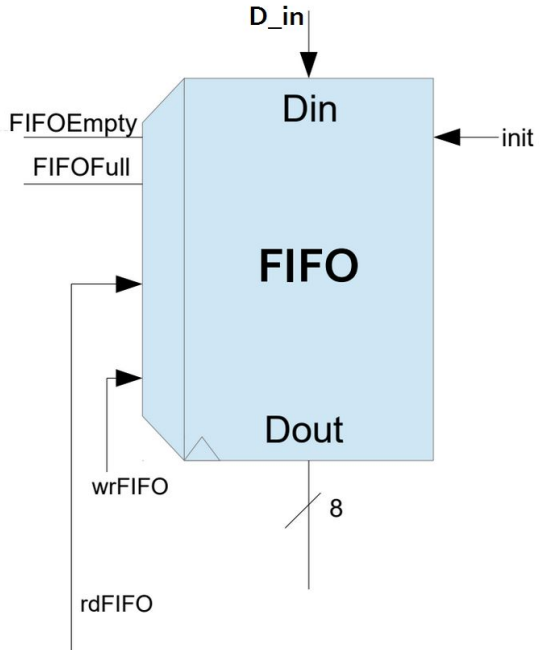
Le registre de status est en accès *R only*, l'utilisation de registre est donc inutile. Seul le registre de demande d'interruption est pertinent car son niveau devra être gardé en mémoire jusqu'à sa lecture. Il est mis à '1' à la fin d'une transmission par l'UC et est remis à zéro lors d'une lecture du registre par *RdOut*. Le registre ***ELNStatus*** est composé des signaux *Busy* et *IRQ* fournis par l'UC, et *FIFOEmpty* et *FIFOFull*, lus directement sur la pile FIFO.




## c) FIFO

Ce composant, qui est fourni, est une pile venant charger la valeur lue sur le bus Avalon ***D_in*** à chaque demande d'écriture *wrFIFO,* créé par le décodeur de l'interface Avalon (cf. *d) Interface Avalon*).

La lecture des données présentes sera contrôlé par l'UC.



## d) Interface Avalon et Décodeur

Afin de décoder quels signaux mettre à '1' en fonction de l'***addr*** et des signaux ***rd*** et ***wr*** Avalon, j'utilise un décodeur combinatoire permettant de différencier les accès lecture ou écriture des différents registres.


Il est défini par les équations :

Lecture des registres :

- *RdOut* = 1 si ***Addr*** = 0 et ***rd*** = 1
  sinon 0

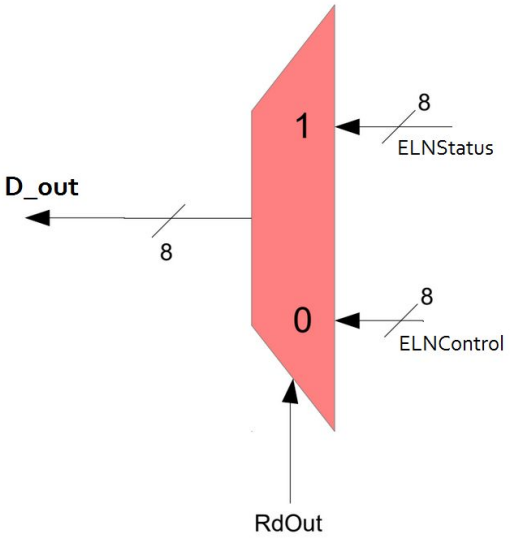
Chargement de ***ELNControl*** dans les registres :

- *ldCtrl* = 1 si ***Addr*** = 1 et ***wr*** = 1
  sinon 0

Écriture d'une donnée dans la pile FIFO :

- *wrFIFO* = 1 si ***Addr*** = 2 et ***wr*** = 1
  sinon 0

Le signal *RdOut* contrôle également un multiplexeur dont le rôle est d'écrire sur le bus Avalon ***D_out*** soit ***ELNStatus*** soit ***ELNControl***.

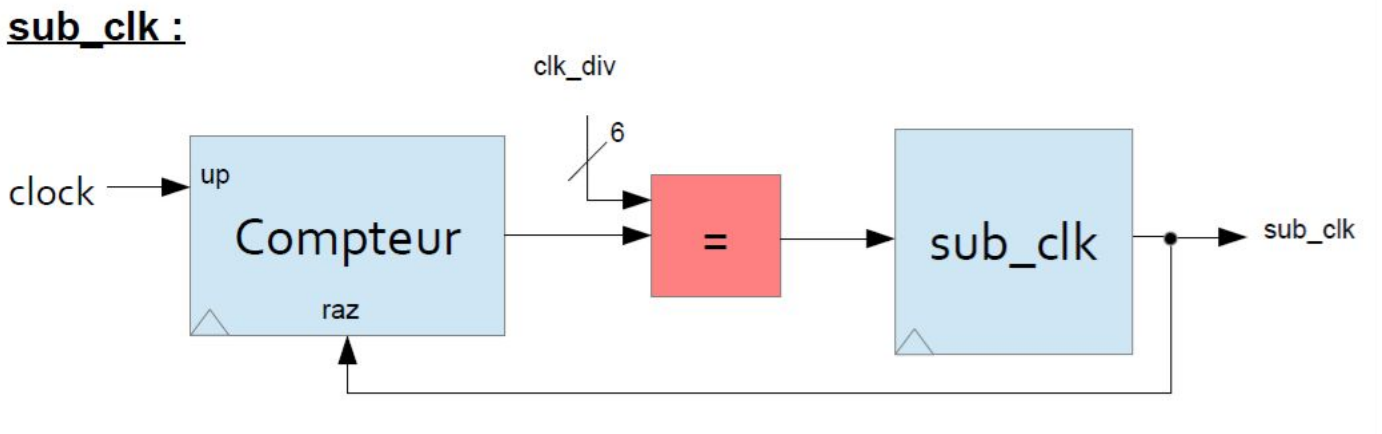



Ainsi, par défaut, le bus Avalon ***D_out*** présente le registre ***ELNControl*** et quand *RdOut* passe à '1', ***ELNStatus*** est lu sur ***D_out*** en remettant le flag d'interruption (*IRQF*) à '0'.

## e) Diviseur de fréquence d'Horloge

Afin de faire fonctionner des blocs synchrones à des fréquences plus lentes pour obtenir la fréquence de fonctionnement de la sortie ***Line***, je fais le choix d'utiliser un diviseur de fréquence. Il génèrera une impulsion toutes les *clk_div* coups d'horloge grâce à un compteur, dont la valeur sera comparée avec le contenu du registre *clk_div*. Ainsi, le changement de valeur de ce dernier change la fréquence de *sub_clock*.

Comme des opérations combinatoires sont effectuées sur un signal qui sera injecté comme signal d'horloge, il est nécessaire qu'il soit filtré par un registre s'assurant de l'absence de glitch.

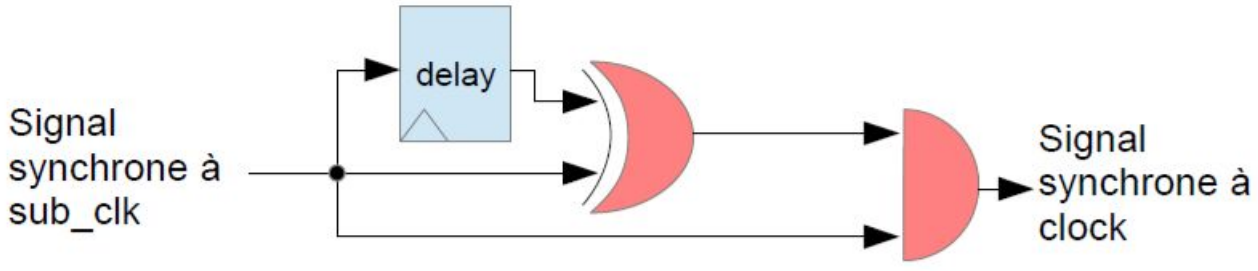



De par la nature de *sub_clk* et la capacité du registre *clk_div*, *sub_clk* pourra varier de $f_{clk}/2$ à $f_{clk}/63$. Il faudra donc s'assurer lors du développement de l'API du composant d'interdire les valeurs critiques telles que

*clk_div* = {0; 1; >63}.

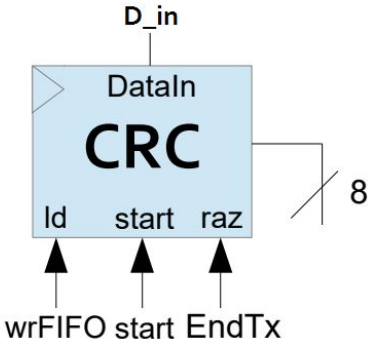
## f) Trigger

La pile FIFO et l'UC fonctionnent avec deux fréquences d'horloges différentes et communiquent par le biais de *rdFIFO*. Il faut adapter ce signal par l'utilisation d'un trigger qui émettra une impulsion unique de largeur 1 *"clock"* quand il reçoit un signal. Sa sortie est à l'état haut quand il détecte un changement de niveau et que le signal d'entrée est à l'état haut. Son fonctionnement peut être rapproché à celui d'un système monostable.



## g) Le CRC (Cyclic Redundancy Check)

À la fin de l'envoi des données, l'émetteur doit transmettre le CRC associé. Pour ce faire, il charge les données en même temps que la pile FIFO pour préparer le calcul. Lorsqu'il reçoit le bit de start qui démarre l'émission, il effectue le calcul avec le polynôme d'ordre 8 : $1 + x^8$ c'est-à-dire "100000001". Une fois le calcul terminé, il présente la valeur du CRC prête à être sérialisée.
Il est remis à zéro à la fin d'une transmission.




Il peut être remarqué que le choix de ce polynôme est particulier. En effet, comme seuls ses premier et dernier bit sont à '1', sa valeur au bout de 8 coups d'horloge sera le résultat d'un XOR entre la donnée injectée et la dernière valeur calculée. Ceci est dû au fait que, seul le dernier bit du polynôme effectuant l'opération, un XOR en bit par bit, du poids fort vers le poids faible, sera fait entre les deux données. Ainsi, si 4 octets O_1, O_2, O_3, et O_4 sont transmis, le CRC obtenu sera :

(((O_1 XOR O_2 ) XOR O_3 ) XOR O_4)

Cependant, dans l'optique de développer un composant facilement réutilisable et adaptable, j'ai fais le choix d'utiliser la méthode de calcul standard, permettant ainsi plus tard si je le souhaite de modifier la valeur du polynôme sans à avoir à modifier entièrement la méthode de calcul du CRC.

## h) Entité Serial

Composée d'une unité de contrôle (UC) et d'une unité de traitement (UT), l'entité serial est le composant qui code, sérialise et insère les bits de stuffing quand nécessaire. Elle utilise deux couches de registres afin de pouvoir précharger des données durant une sérialisation : *Data* et *Dec*. En parallèle, des registres *codingIn* et *codingDec* enregistrent si la donnée dans la couche correspondante devra être codée et stuffée ou non.

Un bloc fonction synchrone *codage* viendra garder en mémoire le dernier bit écrit sur la sortie ***Line*** et si le registre à décalage *Dec* lui présente un zéro, il change d'état.

En sortie de ce registre, une porte NAND sur les 4 derniers niveaux de données et du prochain à écrire autorise le décalage de la donnée. Si tous ces niveaux sont à '1', alors le décalage est bloqué et un niveau '0' est inséré en entrée du codage avant la valeur suivante grâce à un multiplexeur. En même temps que le décalage, un compteur est piloté avec les mêmes signaux permettant d'émettre un signal de synchronisation pour l'UC (*serialBusy*) quand le décalage est en cours.

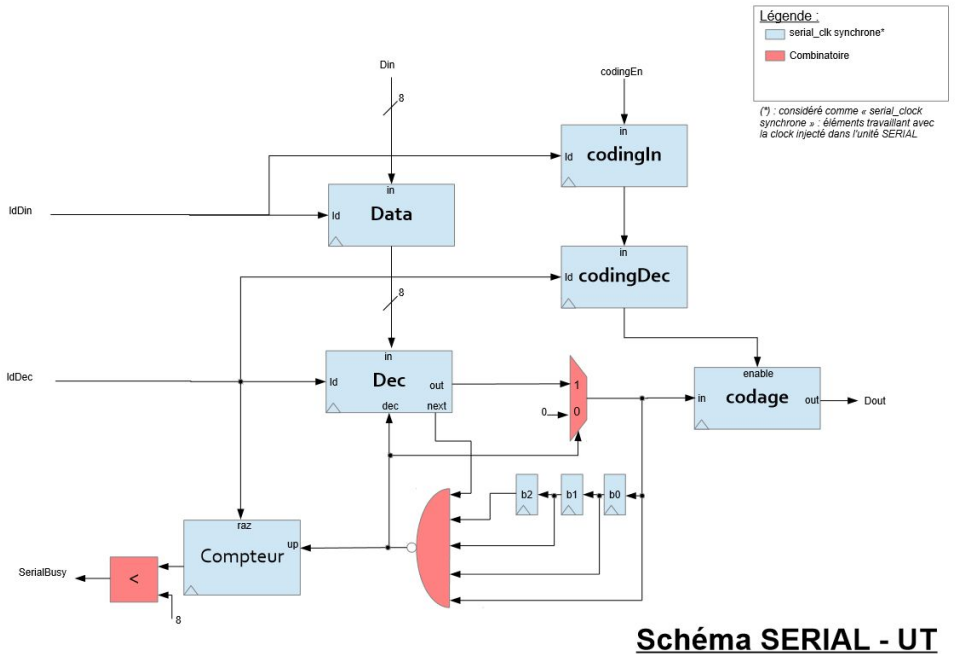



**Schéma SERIAL - UT**

D'un point de vue de l'UC, il s'agit juste de faire l'interface entre la demande de données à sérialiser (*DRq*), la présence de données valides en entrée de l'entité (*DAck*) et la capacité de l'UT à recevoir une nouvelle donnée.

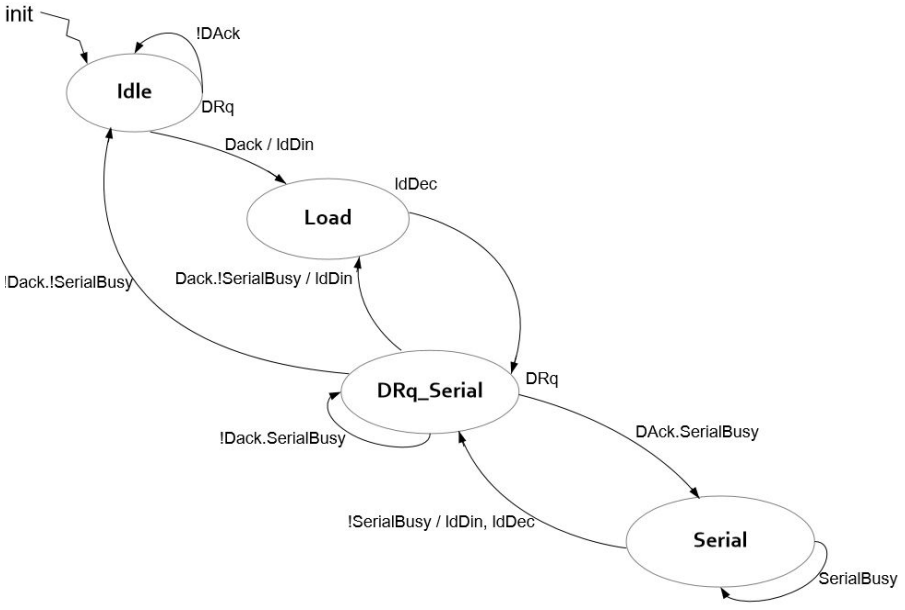
Quatre états sont définis : l'état d'attente **Idle**, Le chargement des registres **Load**, la demande d'une donnée quand la première couche est chargée avec le démarrage de la sérialisation **DRq_Serial**, et la sérialisation **Serial**.

L'UC reste dans l'état **Idle** en demandant une donnée tant que rien ne lui est présenté à sérialiser.

Si cela devient le cas, elle passe dans l'état intermédiaire **Load** afin de charger la donnée à travers les 2 couches de registres de l'UT et se retrouve dans l'état **DRq_Serial**.

Si une nouvelle donnée est présentée alors que la sérialisation n'est pas finie, l'UC passe dans un état d'attente **Serial** le temps de finir la sérialisation puis demande un chargement des registres.

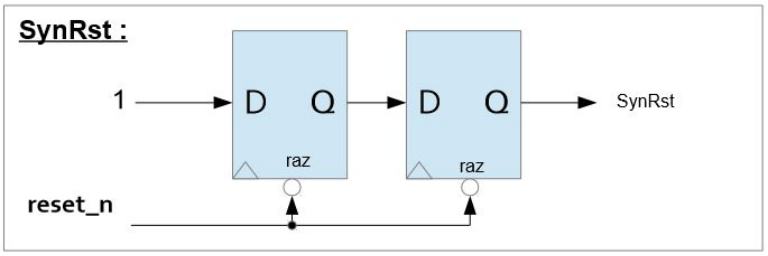
Dans le cas où la sérialisation de la donnée a fini, L'UC revient soit à l'état intermédiaire **Load** si une nouvelle donnée est présente soit vers **Idle** si aucune sérialisation n'est demandée.




**Graphes des états SERIAL - UC**

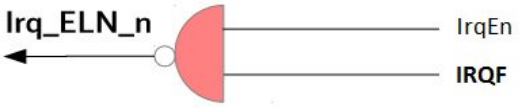
## i) Signal Reset

Afin d'adapter le signal ***Reset_n*** asynchrone en signal utilisable synchrone, j'utilise le schéma suivant qui génère une activation immédiate asynchrone et une désactivation synchrone afin d'empêcher un état métastable.




## j) Sortie d'interruption

La sortie d'interruption ***Irq_ELN_n*** ne peut être active que si le registre *IrqEn* est à '1' et ce-dernier ne doit pas bloquer le flag d'interruption présent sur ***ELNStatus***. Une porte NAND est donc appliquée après le registre *IRQF* qui contient ce flag. Ainsi, quand le flag et *IrqEn* sont à '1', la sortie ***Irq_ELN_n*** bascule bien en actif à l'état bas.




## k) Schéma complet

Maintenant que tous les éléments individuels ont été définis, je peux les assembler pour obtenir la structure globale.

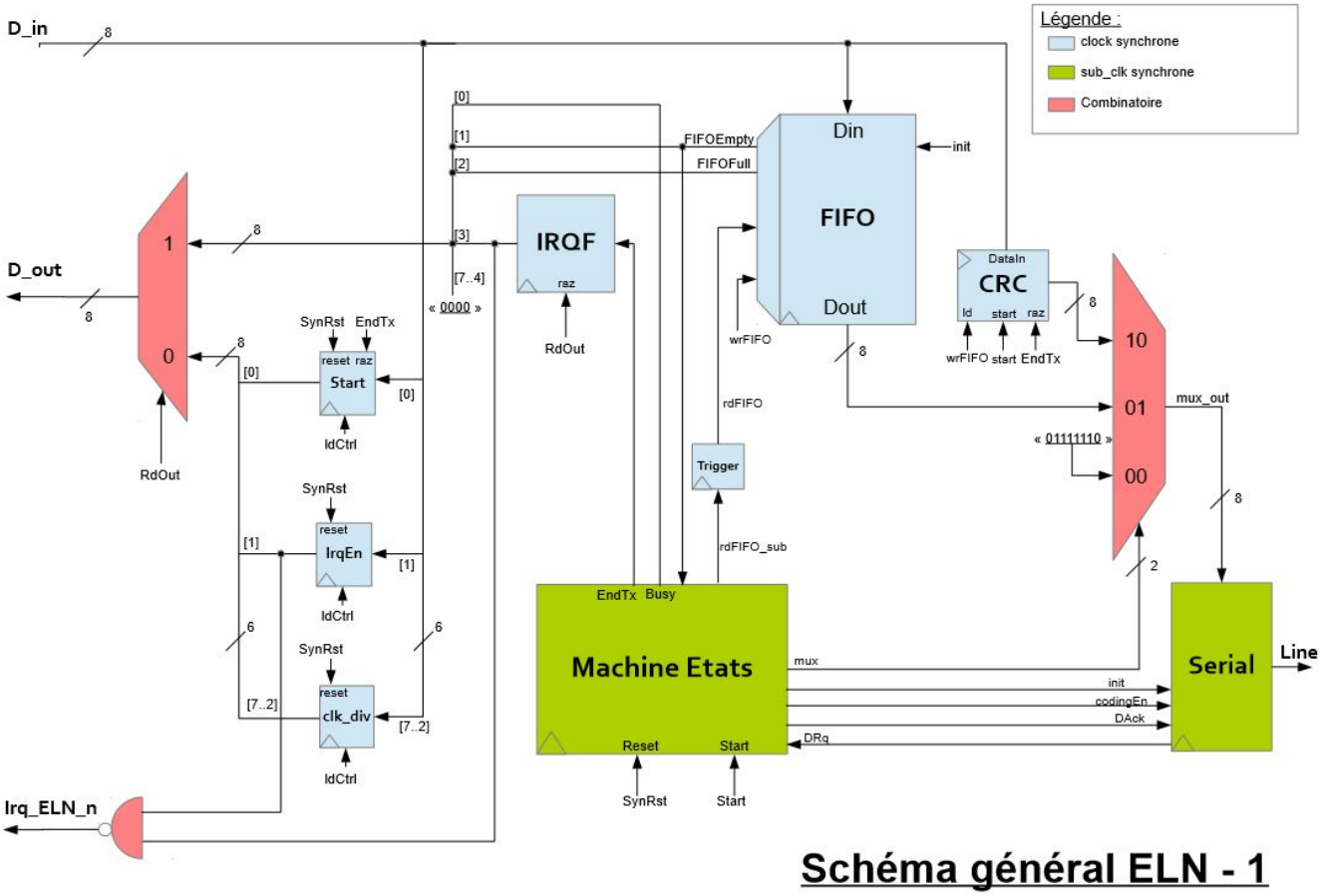
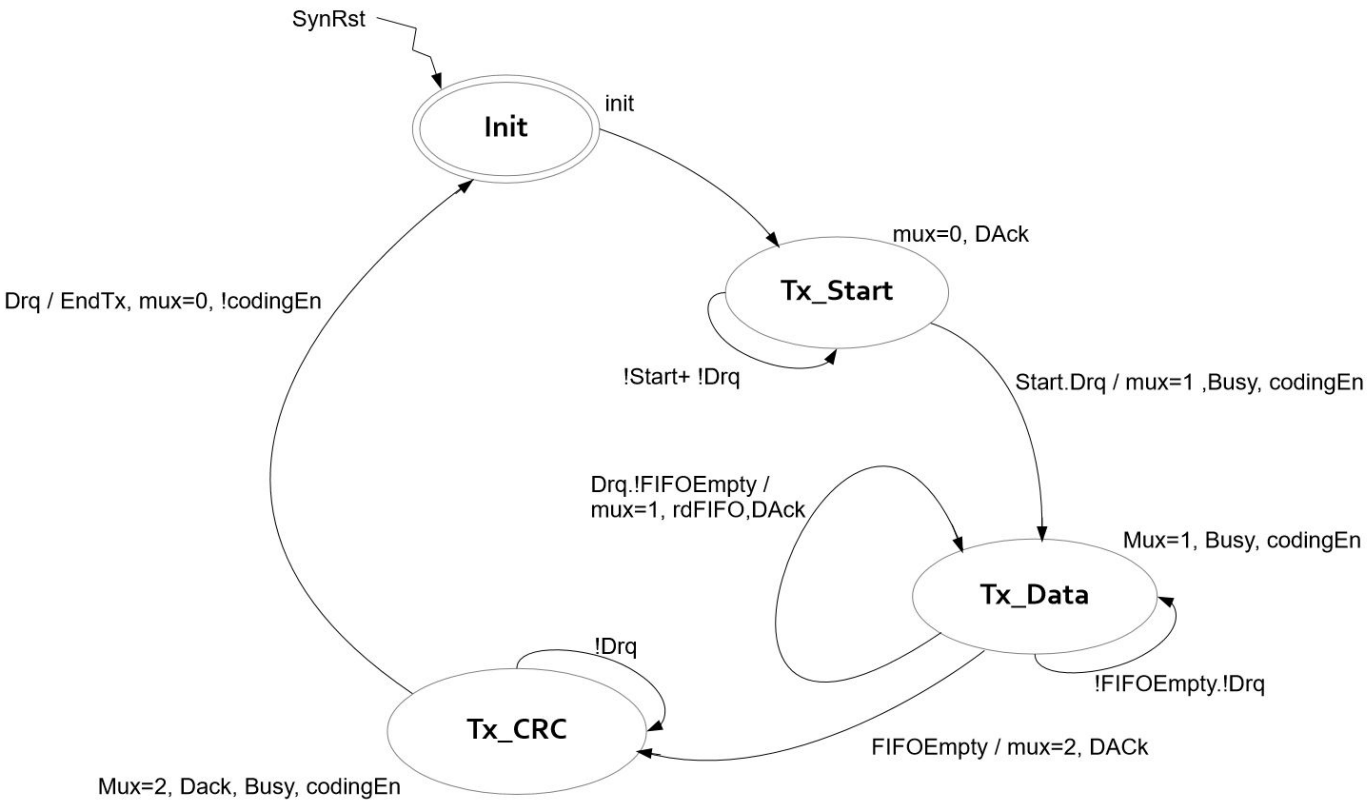



**Schéma général ELN - 1**

## l) Machine à états générale

Au centre du schéma précédent se trouve l'unité de contrôle principale qui est une machine à états gérant l'organisation de la sérialisation, de la lecture de la pile FIFO, et des demandes d'interruption.




A la réinitialisation du composant par le signal *SynRst*, l'UC est en état **Init** où elle demande l'initialisation du système.

Elle passe ensuite sans condition sur l'état de transmission des fanions de start (*mux*=0) **Tx_Start** jusqu'à que le registre de start devienne actif.

Quand une transmission est demandée, l'UC passe en état de transmission de donnée **Tx_Data** et lis les données présentes dans la pile FIFO (*mux*=1) jusqu'à que celle-ci soit vide. Le bit ***Busy*** passe également à 1 pour signifier qu'autre chose qu'un fanion de start est actuellement transmis sur ***Line***.

Quand la pile est vide, le CRC est transmis durant l'état **Tx_CRC**, puis l'UC revient sur l'état **Init** en signalant la fin de de la transmission (*EndTx*), ce qui remet le registre de start à '0' et active le flag d'interruption *IRQF*.

# III) Codage des éléments

Les fichiers de code complet sont fournis en annexe II.

## a) Serial

Le premier élément à coder est la sous-entité Serial qui se charge de sérialiser et encoder les données qui lui sont présentées. Afin d'être plus souple pour la suite du développement du composant, j'implémente cette entité de manière à ce que la taille des données reçues soit générique. Dû à cette généricité, un bit de synchronisation indiquant la fin du décalage du registre est obligatoire, c'est le signal *serialBusy*.

J'implémente donc d'abord le fonctionnement de l'UT Serial présentée précédemment.

On notera que les registres *Data* et *Dec* sont initialisés à 0 afin d'éviter une sérialisation et des opérations combinatoires sur des états "*undefined*".

```
architecture rtl of UT_SerialData is
	signal Data, Dec : std_logic_vector(DATA_WIDTH-1 downto 0) := (others=>'0');
	signal out_b, State, lastState, codingIn, codingDec : std_logic ;
	signal Delay_b0, Delay_b1, Delay_b2, next_b : std_logic;

begin
	Registres : process(Clk)
	variable stuffing : std_logic;
	variable counter : integer range 0 to DATA_WIDTH;
	begin
		if rising_edge(Clk) then


			stuffing := Delay_b2 and Delay_b1 and Delay_b0 and out_b and next_b;

			-- update delays registers
			Delay_b2 <= Delay_b1;
			Delay_b1 <= Delay_b0;
			Delay_b0 <= out_b;

			if LdDIn = '1' then -- load Din
				Data <= DIn;
				codingIn <= codingEn;
			end if;

			if LdDec = '1' then -- load Dec and first shift
				counter := 0;
				codingDec <= codingIn;

				if stuffing = '1' and codingIn = '1' then -- if five '1' were transmitted while coding enabled
					Dec(DATA_WIDTH-1 downto 0) <= Data(DATA_WIDTH-1 downto 0);
					out_b <= '0'; -- write 0
					stuffing := '0';
				else -- shift
					Dec(DATA_WIDTH-2 downto 0) <= Data(DATA_WIDTH-1 downto 1);
					Dec(DATA_WIDTH-1) <= '0';
					counter := counter + 1;
					out_b <= Data(0);
					next_b <= Data(1);
				end if;
			else

				if stuffing = '1' and codingDec ='1' then -- if five '1' were transmitted while coding enabled
					out_b <= '0'; -- write 0
					stuffing := '0';
				else -- shift
					Dec(DATA_WIDTH-2 downto 0) <= Dec(DATA_WIDTH-1 downto 1);
					Dec(DATA_WIDTH-1) <= '0';
					counter := counter + 1;
					out_b <= Dec(0);

					if counter < DATA_WIDTH then next_b <= Dec(1);
					else next_b <= Data(0); end if;
				end if;

			end if;
```

```
            if counter < DATA_WIDTH then serialBusy <= '1'; else serialBusy <='0'; end if;

            if (codingDec ='1' or (LdDec = '1' and codingIn = '1')) then
                    State <= not(out_b xor State) ;
            else
                    State <= out_b;
            end if;

            lastState <= State;


            end if;
        end process Registres;

        -- Line out
        Dout <= State;

end architecture rtl;
```

Le code peut paraître compacte et complexe, pourtant, cela est nécessaire car beaucoup de combinaisons différentes de signaux d'entrée sont possibles et il est absolument nécessaire de ne pas perdre de coup d'horloge entre le chargement et le décalage.

L'UC est beaucoup plus simple et son codage est transparent de son graphe. Des sorties de Mealy sont utilisées pour *ldDin* autour de l'état de chargement afin de gagner un coup d'horloge et qu'une donnée soit toujours présente dans la première couche de registre quand la deuxième sera chargée.

```
        RegEtat : process(Clk)
        begin
                if rising_edge(Clk) then
                        if Init = '1' then
                                Etat_cr <= Idle;
                        else
                                Etat_cr <= Etat_sv;
                        end if;
                end if;
        end process RegEtat;

        process(Etat_cr, DAck, serialBusy)
        variable state : std_logic_vector(1 downto 0);
        begin
                DRq <= '0'; LdDIn <= '0'; LdDec <= '0';
                Etat_sv <= Etat_cr;
                state := DAck & serialBusy;
                case Etat_cr is
                        when Idle =>            DRq <= '1';
                                                if DAck = '1' then
                                                        LdDIn <= '1';
                                                        Etat_sv <= Load;
                                                end if;

                        when Load =>            LdDec <= '1';
                                                Etat_sv <= DRq_Serial;

                        when DRq_Serial =>      DRq <= '1';
                                                case state is
                                                        when "10" =>    LdDIn <= '1'; Etat_sv <= Load;
                                                        when "11" =>    Etat_sv <= Serial;
                                                        when "00" =>    Etat_sv <= Idle;
                                                        when others =>  NULL;
                                                end case;

                        when Serial =>          if serialBusy = '0' then
                                                        ldDec<= '1'; LdDIn <= '1';
                                                        Etat_sv <= DRq_Serial;
                                                end if;
                end case;
        end process;
```

## b) Le CRC (Cyclic Redundancy Check)

Dans la même optique que l'entité Serial, le calcul du CRC est implémenté avec une taille de données générique, ce qui complexifie grandement le code. De plus, à cause du choix de l'ordre du polynôme pour le calcul, la généricité n'est pas totale : le polynôme est défini en valeur par défaut dans la déclaration des signaux.

```
signal poly : std_logic_vector(8 downto 0) := "100000001";
```

```
process(Clk, SynRst)
variable level : integer range 0 to (SIZE*DATA_WIDTH);
begin

        if SynRst = '0' then
                CRC_reg <= (others=>'0');
                level := (SIZE*DATA_WIDTH);

        else
                if rising_edge(Clk) then
                        if RAZ = '1' then
                                CRC_reg <= (others=>'0');
                                level := SIZE*DATA_WIDTH;

                        elsif ldCRC = '1' and level > 7 then
                                CRC_reg(level+7 downto level) <= CRC_in;
                                CRC_reg(level-1 downto 0) <= (others=>'0');
                                level := level - 8;

                        elsif start = '1' and level < (SIZE*DATA_WIDTH) then
                                if(CRC_reg((SIZE*DATA_WIDTH)+7) = '1') then -- if MSB = 1
                                        CRC_reg((SIZE*DATA_WIDTH)+7 downto 0) <= (CRC_reg((SIZE*DATA_WIDTH)+6 downto ((SIZE-1)*DATA_WIDTH)+7)
                                                                                xor poly(DATA_WIDTH-1 downto 0))&CRC_reg(((SIZE-1)*DATA_WIDTH)+6 downto 0)&'0';
                                        level := level + 1;

                                else
                                        -- shift
                                        CRC_reg((SIZE*DATA_WIDTH)+7 downto 0) <= CRC_reg((SIZE*DATA_WIDTH)+6 downto 0)&'0';
                                        level := level + 1;
                                end if;

                        end if;
                end if;
        end if;
end process;
CRC_out <=  CRC_reg((SIZE*DATA_WIDTH)+7 downto ((SIZE-1)*DATA_WIDTH)+8);
```

Le code fonctionne en 2 parties :

- *ldCRC* = '1' : la première partie consiste à créer un signal contenant les données visant à être transmises sur la ligne. Il est important de connaître la taille de la trame pour effectuer le bon nombre de cycles de calculs. Le nombre maximum de données à assembler doit correspondre à la capacité maximale de la pile.

- *Start* = '1' : La deuxième partie est le décalage du signal précédemment créé et un XOR avec le polynôme quand nécessaire. La variable *level* contient le rang des bits utiles au calcul. Ainsi, si d'autres données sont chargées alors que le calcul à commencé, elles seront ajoutées à la suite.

Le résultat du calcul est présenté en sortie du bloc et est prêt à être sérialisé.

La raison pour laquel la taille du calcul du CRC est précisé et que le niveau de la pile FIFO n'est pas simplement regardé est, qu'afin d'effectuer les décalages sans être perturbé, l'algorithme doit gérer ces valeurs indépendamment du reste.

## c) Machine à états générale

La machine à états de l'entité de plus haut niveau est relativement simple et correspond à son graphe. Par défaut, la commande du *mux* vaut "00", ce qui correspond à la sérialisation de fanions de start.

```
RegStates : process(Clk, SynRst)
begin
        if SynRst = '0' then
                        State_cr <= s_Init;
        elsif rising_edge(Clk) then
                State_cr <= State_sv;
        end if;
end process RegStates;

process(State_cr, Start, DRq, FIFOEmpty)
begin
        init <= '0'; DAck <= '0'; rdFIFO <= '0'; EndTx <= '0'; Busy <= '0'; codingEn <= '0'; mux <= "00";

        State_sv <= State_cr;

        case State_cr is
                when s_Init =>  init <= '1';
                                State_sv <= s_Tx_Start;

                when s_Tx_Start =>
                                mux <= "00"; DAck <= '1';
                                if Start = '1' and DRq = '1' then
                                        State_sv <= s_Tx_Data;
                                        Busy <= '1';
                                        mux <= "01";
                                        codingEn <= '1';
                                end if;

                when s_Tx_Data =>
                                mux <= "01"; Busy <= '1'; codingEn <= '1';
                                if FIFOEmpty = '1' then
                                        mux <= "10";
                                        DAck <= '1';
                                        State_sv <= s_Tx_CRC;
                                elsif DRq = '1' then
                                        mux <= "01";
                                        rdFIFO <= '1';
                                        DAck <= '1';
                                end if;

                when s_Tx_CRC =>
                                mux <= "10"; DAck <= '1'; Busy <= '1'; codingEn <= '1';
                                if DRq = '1' then
                                        State_sv <= s_Tx_Start;
                                        EndTx <= '1';
                                        mux <= "00";
                                        codingEn <= '0';
                                end if;

        end case;
end process;
```

## d) Ensemble & autres composants

Par simplicité, j'implémente les autres blocs fonctionnels, registres et entités dans l'entité de plus haut niveau.

Je commence par les registres :

```
-- Start

	process(Clk, SynRst)
	begin
		if SynRst = '0' then
			start <= '0';
		else
			if rising_edge(Clk) then
				if EndTx = '1' then
					start <= '0';
				elsif ldCtrl = '1' then
					start <= D_in(0);
				end if;
			end if;
		end if;
	end process;
```

À la suite du registre *IrqEn*, je mets toute la partie qui concerne les demandes d'interruption, tel que le registre du flag ou la porte NAND comme présentés précédemment.

```
-- IRQEn

	process(Clk, SynRst)
	begin
		if SynRst = '0' then
			IRQEn <= '0';
		else
			if rising_edge(Clk) then
				if ldCtrl = '1' then
					IRQEn <= D_in(1);
				end if;
			end if;
		end if;
	end process;

-- IRQF

	process(Clk, SynRst)
	begin
		if SynRst = '0' then
			int_irq <= '0';
		else
			if rising_edge(Clk) then
				if RdOut = '1' then
					int_irq <= '0';
				elsif EndTx = '1' then
					int_irq <= '1';
				end if;
			end if;
		end if;
	end process;

-- IRQ
	IRQ_ELN_n <= not(int_irq and IRQEn);
```

En code concurrent, je peux également déclarer les différents MUXs du schéma :

```
-- MUX
        D_out <=        clk_div&IRQEn&start when RdOut = '0' else
                        "0000"&int_irq&FIFOFull&FIFOEmpty&Busy;

        mux_out <=      FIFOout when mux = "01" else
                        CRC_out when mux = "10" else
                        "01111110";
```

Ainsi que le décodeur de l'interface Avalon :

```
-- Decodeur

        ldCtrl <= '1' when Addr = "01" and Wr='1'
        else '0';

        wrFIFO <= '1' when Addr = "10" and Wr='1'
        else '0';

        RdOut <= '1' when Addr = "00" and Rd='1'
        else '0';
```

Le code qui génère *sub_clk* à partir de *clk_div* étant synchrone dû au registre en sortie, il est nécessaire de l'implémenter dans un process. Par simplicité et cohérence, j'intègre au process également la gestion du registre *clk_div*. Je retrouve bien la valeur par défaut "001000" (8).

```
-- Clock Divisor

        process(Clk, SynRst)
        variable counter : integer range 0 to 63;
        begin
                if SynRst = '0' then
                        clk_div <= "001000";
                        counter := 0;
                        sub_Clk <='0';

                elsif rising_edge(Clk) then
                        if ldCtrl = '1' then
                                clk_div <= D_in(7 downto 2);
                        end if;

                        counter := counter + 1;

                        if counter >= to_integer(unsigned(clk_div)) then
                                sub_Clk <='1';
                                counter := 0;
                        else
                                sub_Clk <='0';
                        end if;

                end if;

        end process;
```

Il faut ensuite définir le bloc trigger permettant d'adapter le signal communiquant à des fréquences d'horloges différentes :
(*rdFIFO_sub* correspond au signal *rdFIFO* émis par l'UC, donc synchrone à *sub_clk* )

```
-- signal sub_clock interface

        process(Clk)
        begin
                if rising_edge(Clk) then
                                rdFIFO_delay <= rdFIFO_sub;
                                rdFIFO <= (rdFIFO_sub xor rdFIFO_delay) and rdFIFO_sub;
                        end if;
        end process;
```

La génération d'un signal à désactivation synchrone créé à partir de ***Reset_n*** se fait suivant le schéma présenté en partie **II)** :

```
-- Synchronous Reset

        process(Clk, reset_n)
        begin
                if reset_n = '0' then
                        t_rst <= '0'; SynRst <= '0';
                elsif rising_edge(Clk) then
                        t_rst <= '1';
                        SynRst <= t_rst;
                end if;
        end process;
```

Pour finir, j'instancie les entités externes dans la même architecture en précisant les paramètres génériques :
Le cahier des charges propose une taille de pile FIFO de 4 octets, soit le paramètre FIFO_SIZE = $\log_2(4) = 2$

```
serial :        entity SerialData
                        generic map (   DATA_WIDTH => 8)
                        port map(       Clk => sub_Clk,
                                        Init => init,
                                        DAck => DAck,
                                        codingEn => codingEn,
                                        DIn => mux_out,
                                        DOut => Line_ELN,
                                        DRq => DRq
                                        );

machineEtat :   entity State_machine
                        port map(       Clk => sub_Clk,
                                        SynRst => SynRst,
                                        init => init,
                                        Start => Start,
                                        DRq => DRq,
                                        FIFOEmpty => FIFOEmpty,
                                        DAck => DAck,
                                        rdFIFO => rdFIFO_sub,
                                        EndTx => EndTx,
                                        Busy => Busy,
                                        codingEn => codingEn,
                                        mux => mux
                                        );
```

```
parite : entity CRC
                        generic map (   DATA_WIDTH => 8,
                                        SIZE => 4
                                        )
                        port map(       Clk => Clk,
                                        SynRst => SynRst,
                                        RAZ => endTx,
                                        start => Start,
                                        ldCRC => wrFIFO,
                                        CRC_in =>  D_in,
                                        CRC_out => CRC_out
                                );

fifo :  entity FIFO_nMots_mBits
                        generic map (   DATA_WIDTH => 8,
                                        FIFO_SIZE => 2
                                        )
                        port map(       Horloge => Clk,
                                        initFifo => init,
                                        WrFifo => wrFIFO,
                                        RdFifo => rdFIFO,
                                        DataIn => D_in,
                                        DataOut => FIFOout,
                                        FifoEmpty => FIFOEmpty,
                                        FifoFull => FIFOFull
                                        );
```

et je déclare les ports du composant qui correspondent bien à la définition effectuée lors de la présentation du cahier des charges :

```
entity eln is
        port(   Clk, reset_n, rd, wr : in std_logic;
                Line_ELN, IRQ_ELN_n : out std_logic;
                D_in : in std_logic_vector(7 downto 0);
                D_out : out std_logic_vector(7 downto 0);
                Addr : in std_logic_vector(1 downto 0)
        );
end entity eln;
```

# IV) Simulation fonctionnelle

Maintenant que le composant est intégralement implémenté, il s'agit de vérifier qu'il fonctionne comme je le souhaite. Je testerai les parties principales qui sont l'interface Avalon, l'UC de plus haut niveau, le calcul du CRC et la sous-division d'horloge. Si ces-derniers sont validés, alors je testerai la sérialisation des données, ce qui correspondra à un test d'ensemble du composant.

## a) Interface Avalon

Le premier test à effectuer est celui de l'interface Avalon pour avoir une communication avec le processeur NIOSII. Le test consiste à venir écrire des données et lire les registres à plusieurs moments afin de voir leurs évolutions.

Ici, je décide de tester :

- Lecture status
- Lecture control
- Écriture de 2 données ('a' & 'b')
- Lecture status
- Écriture de 2 données ('c' & 'd')
- Lecture status
- Écriture control (clk_div = 2 | start = 1)
- Lecture control
- Lecture status

Dans le but de lancer des simulations plus facilement, j'utilise des macros TCL.

Après avoir défini les adresses des registres :

```
set ELNStatus 0
set ELNControl 1
set ELNFIFO 2
```

les tailles des bus :

```
set BusAddressSize 2
set BusDataSize 8
```

les polarités des signaux :

```
set NiveauActif    1
set NiveauActifReset 0
set DureeReset 2.5
```

et leur noms :

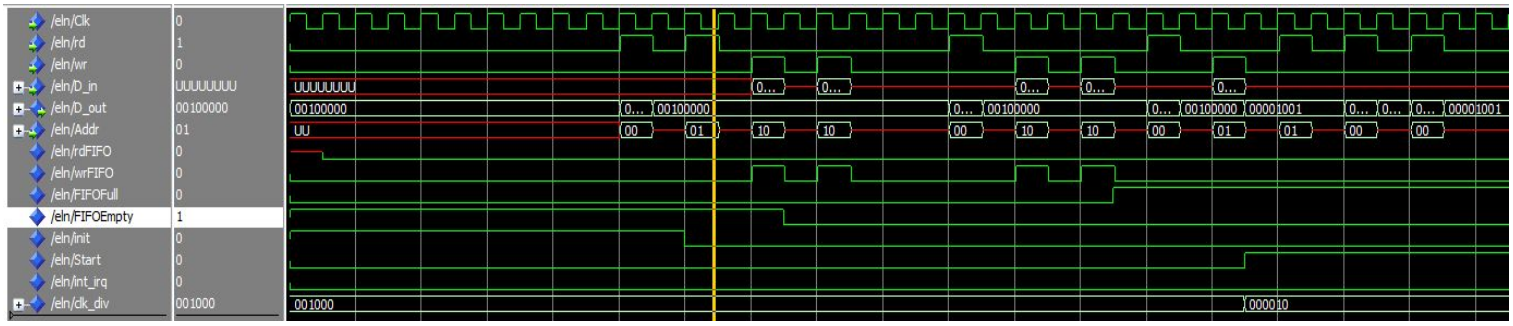
```
set HorlogePeripherique Clk
set ResetPeripherique reset_n
set AdresseRegistrePeripherique addr
set AccesLecturePeripherique rd
set AccesEcriturePeripherique wr
set BusEcritureDonneesPeripherique D_in
set BusLectureDonneesPeripherique D_out
set InterruptRequest IRQ_ELN_n
```

Je peux ainsi traduire le scénario précédent par :

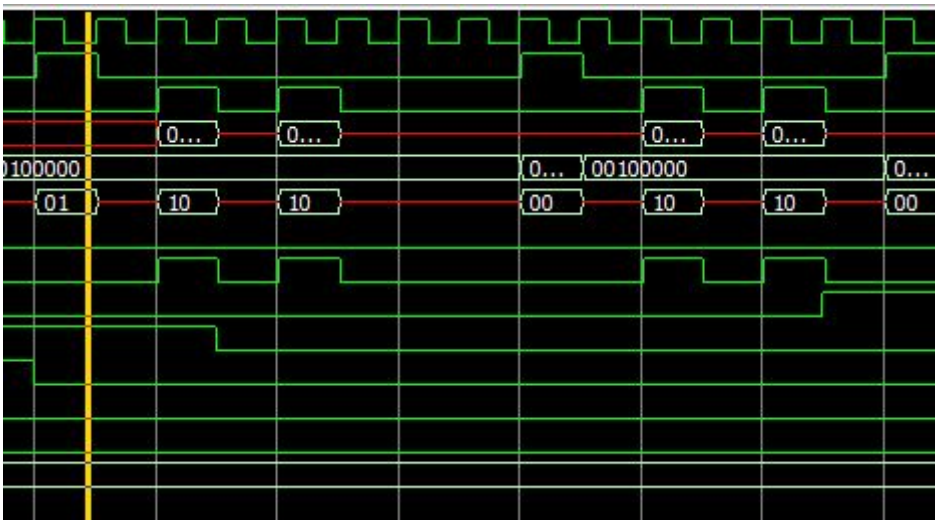
```
set CyclesAvalon {
      {10 ELNStatus }
      {12 ELNControl }
      {14 ELNFIFO 16#61}
      {16 ELNFIFO 16#62}
      {20 ELNStatus }
      {22 ELNFIFO 16#63}
      {24 ELNFIFO 16#64}
      {26 ELNStatus }
      {28 ELNControl 2#0001001}
      {30 ELNControl }
      {32 ELNStatus }
}
```

(Par souci de place et de facilité de lecture, seuls les scripts en langage formel seront donnés par la suite )

J'obtiens :




En zoomant sur une partie :

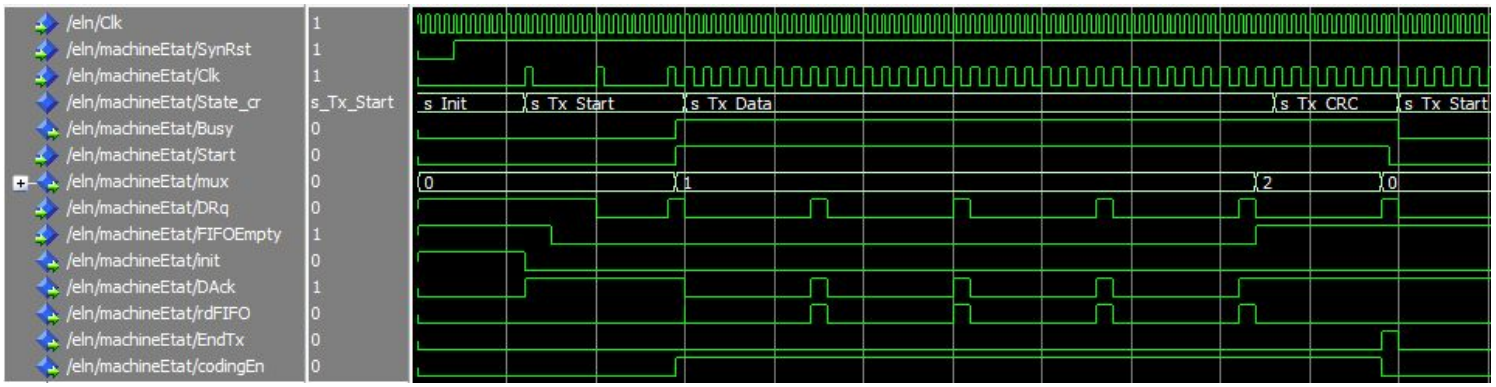


J'observe que la lecture des registres renvoie les valeurs demandées et qu'une écriture dans un des paramètres de *ELNControl* est bien sauvegardé, tel que *clk_div* par exemple (dernière ligne). Je constate également que le registre *ELNStatus* s'actualise en fonction de l'état de la pile FIFO avec les paramètres Empty et Full. Ceci permet donc de valider les registres, le décodeur et la pile FIFO.

Ainsi, je sais que non seulement l'interconnexion Avalon marche correctement et de plus, je peux m'en servir pour les tests suivants, ce pourquoi cela a été ma première partie testée.

## b) Machine à Etats générale

Le point le plus important du composant est la machine à états de l'Unité de Contrôle principale. C'est elle qui régit entièrement le fonctionnement et l'ordre des actions effectuées par les autres composants. En utilisant le script précédent et en admettant que l'entité de sérialisation communique de manière adéquate (je me pencherai sur son bon fonctionnement plus tard), je vérifie le parcours des états et le niveau des sorties qui sont contrôlées par l'UC.




J'observe que la machine à états passe dans l'état **Init** à l'activation du signal *SynRst*, puis en **Tx_Start** en attente du bit *start* à '1'. Quand une transmission est demandée, elle active l'état **Tx_Data** jusqu'à ce que la pile FIFO soit vide, reste 8 coups d'horloge sur **Tx_CRC** et revient en état **Tx_Start**. L'état *Busy* est bien à '1' quand l'UC demande une sérialisation, soit d'une donnée, soit du CRC.

Je peux déjà constater que le diviseur d'horloge à changé de fréquence de fonctionnement lors de l'écriture sur ***ELNControl*** de "00001001" (`clk_div = 2 | start = 1`).

## c) Le CRC (Cyclic Redundancy Check)

Le test du calcul du CRC doit non seulement attester de son bon fonctionnement dans le cas standard d'une pile pleine autant que dans le cas d'un nombre variable d'éléments.

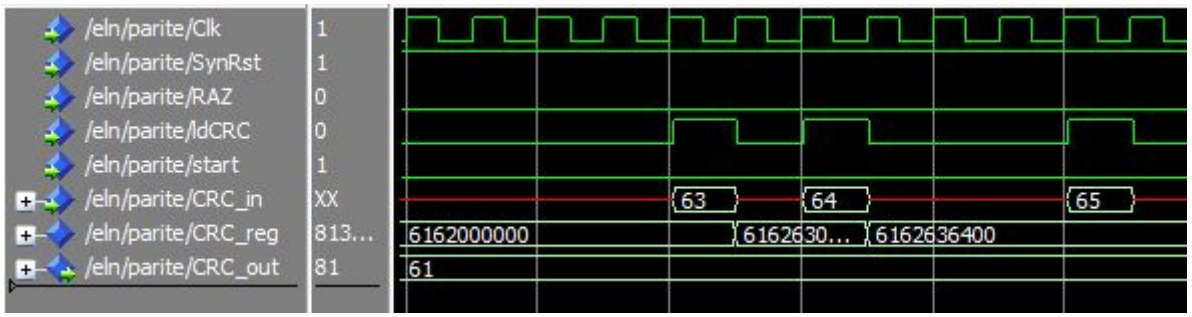
Le site suivant permet de vérifier facilement les valeur de CRC.

Ainsi, en écrivant la suite "0x61 0x62 0x63 0x64" je devrais m'attendre en fin de calcul à un CRC valant 0x04.

Je vérifie d'abord le bon fonctionnement de la création du registre *CRC_reg* :


J'ai bien le registre *CRC_reg* présentant la valeur 0x6162636400, prêt à être calculé.

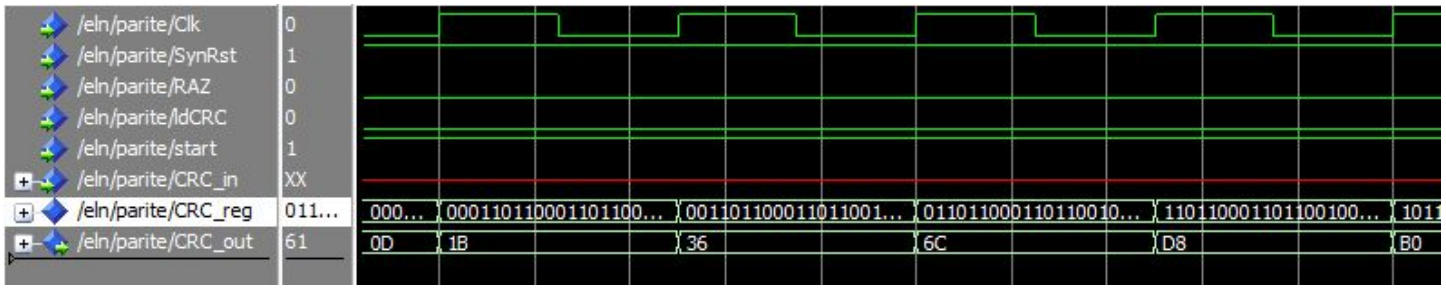
Si j'essaie d'introduire une 5ième valeur supérieure à la capacité de la pile et donc à la capacité de calcul du CRC, elle n'est pas prise en compte.




*CRC_reg* ne change pas de valeur alors que le chargement de 0x65 à été demandé.

Je vérifie maintenant que la valeur se décale correctement dans le registre *CRC_reg*.

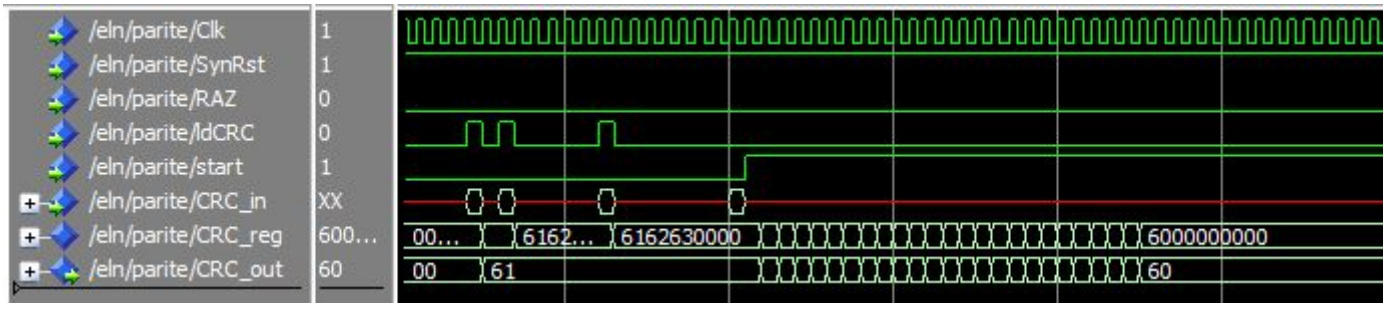
En zoomant sur la phase de calcul, j'observe le bon déroulement de l'algorithme de calcul du CRC (décalage et XOR quand nécessaire).




Et ainsi, en fin de calcul, j'obtiens bien la valeur désirée : 0x04.

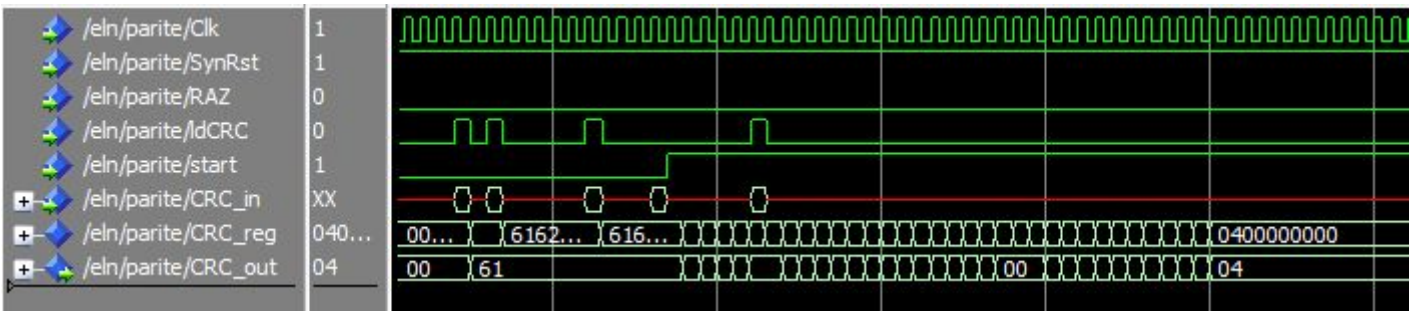

Le calcul doit aussi se dérouler correctement même si l'utilisateur demande une sérialisation alors que la pile n'est pas pleine.

En n'insérant que trois données 0x61 0x62 0x63, j'attends un résultat à 0x60.




Je vérifie pour finir que si une donnée est chargé alors qu'une sérialisation est en cours, comme celle-ci sera prise en compte par la pile FIFO, il faut également que le calcul du CRC soit modifié en conséquence.

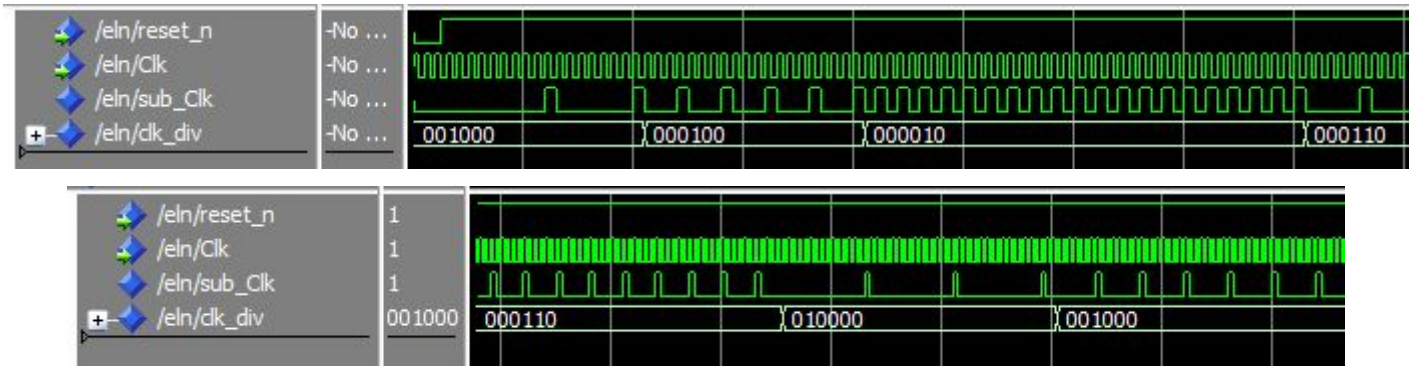
Je charge 0x61 0x62 et 0x63, je lance le calcul et demande le chargement de 0x64.




En chargeant la dernière donnée (0x64) après le début du calcul, celle-ci est bien chargé dans *CRC_reg* et le résultat revient bien au même (0x04).
Après ce dernier test, je peux considérer que le bloc de calcul du CRC est fonctionnel et validé.

## d) Diviseur de fréquence d'Horloge

Ce test est relativement simple pourtant très important car une génération de signal d'horloge doit être exempt de tous "glitches" . Je change donc la valeur de *clk_div* en début, en cours, et en fin de comptage pour vérifier qu'il n'y ait jamais d'erreurs ou d'arrêt de *sub_clk*. Seul après une activation de ***reset_n***, un vide de *clk_div* coup d'horloge est présent car je considère que les système auront changé d'état lors de la réinitialisation et donc il faut attendre avant le prochain coup d'horloge.




## e) Sérialisation & Ensemble

Je peux désormais tester l'ensemble afin de regarder le comportement de l'entité de sérialisation et le niveau écrit sur la sortie ligne.
Je choisis le scénario de sérialiser les caractères 'a', 'b', 'c', et 'd' soit 0x61, 0x62, 0x63, et 0x64.
Le CRC sera donc 0x04.

En langage binaire :

```
01100001 01100010 01100011 01100100 00000100
```

Sur la ligne, les bits de poids faibles sont envoyés en premiers. Sans codage, la trame est la suivante:

```
10000110 01000110 11000110 00100110 00100000
```

Il faut maintenant la coder par changement d'état en fonction du bit à transmettre :

```
(0)  01010001 00101110 00101110 10010001 01101010.
```

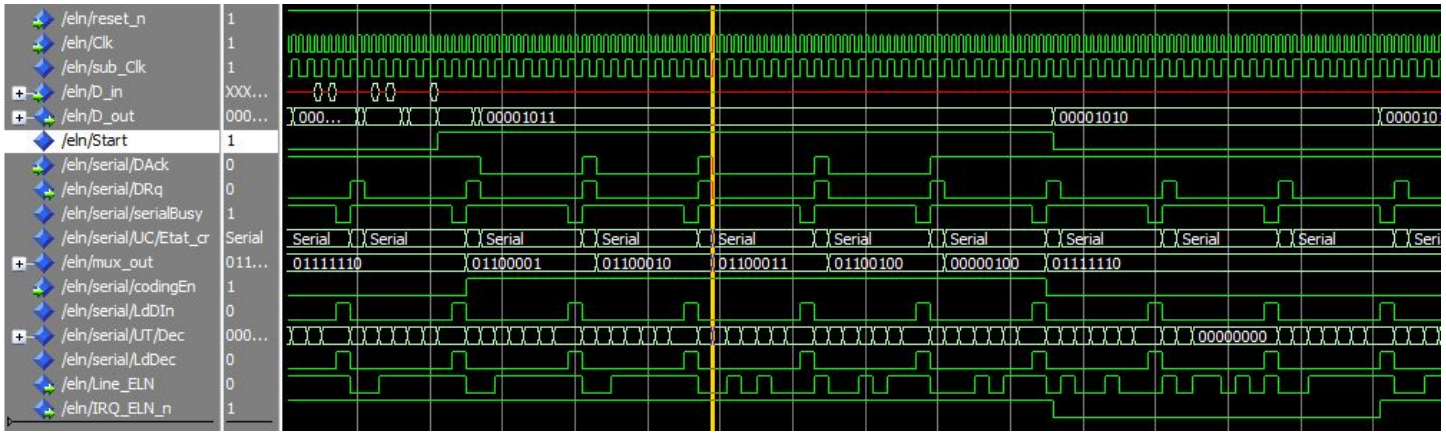
Je m'attends donc à retrouver sur la lignes les niveaux :

| Fanion | 1er octet | 2eme octet | 3eme octet | 4eme octet | CRC |
|---|---|---|---|---|---|
| 01111110 | 01010001 | 00101110 | 00101110 | 10010001 | 01101010 |

J'utilise le scénario suivant:

- Lecture status
- Lecture control
- Écriture de 2 données ('a' & 'b')
- Lecture status
- Écriture de 2 données ('c' & 'd')
- Lecture status
- Écriture control (clk_div = 2| IrqEn = 1 | start = 1)
- Lecture control
- Lecture status
- Lecture status

J'active les interruptions afin de pouvoir visualiser le signal de fin de transmission.




*La transmission des données commence au marqueur.*

Des fanions de start sont bien présents avant et après. Je remarque que 2 fanions sont transmis entre la demande d'envoi *Start* et la réelle transmission de données ce qui correspond au temps de traverser les 2 couches de registres de l'entité de sérialisation.
Quant à ***Line***, les niveaux présents sur la ligne sont bien ceux attendus.

J'observe bien que l'interruption s'active dès que le CRC est transmis et que le composant est prêt pour commencer une nouvelle sérialisation, même si les registres à décalage n'ont pas encore fini de se vider. À ce même moment, le bit de *Start* se réinitialise bien tout seul.
Lors de la première lecture suivante du registre ***ELNStatus***, le flag d'interruption se baisse.

Je peux également voir sur la simulation le comportement de la machine à état de l'entité de sérialisation qui, une fois qu'elle a été chargée, bascule entre les 2 états **DRq_Serial** et **Serial**.

Cependant, ce test représente un cas simple pour vérifier le comportement général des signaux. Il faut maintenant s'assurer que les bits de stuffing nécessaires pour la synchronisation de l'horloge en réception soient placés au bon moment. Pour cela je teste tous les cas possibles.

Les bits de stuffing sont insérés dans la transmission de la donnée entre parenthèses et provoque une altération en rouge sur la ligne ***Line***.

1) <u>stuffing au milieu d'un octet de donnée</u>

Je transmets les données permettant l'apparition d'un bit de stuffing au milieu d'un octet de donnée.

| donnée | 0xFE | 0x2A | 0xBE | 0x5E | 0x34(CRC) |
|---|---|---|---|---|---|
| transmission | 01111(0)111 | 01010100 | 01111(0)101 | 01111010 | 00101100 |
| ***Line*** | 111110000 | 11001101 | 000001100 | 11111001 | 01100010 |


De plus ce test permet de prouver que le bit de stuffing est inséré pour des niveaux haut ou bas de ***Line***.

2) <u>stuffing entre deux octets de donnée</u>

Je transmets les données permettant l'apparition d'un bit de stuffing entre deux octets de donnée.

| donnée | 0xF0 | 0x55 | 0x55 | 0xAA | 0x5A(CRC) |
|---|---|---|---|---|---|
| transmission | 00001111(0) | 10101010 | 10101010 | 01010101 | 01011010 |
| ***Line*** | 101000001 | 10011001 | 10011001 | 00110011 | 00111001 |


3) <u>stuffing avant l'octet de CRC</u>

Je transmets les données permettant l'apparition d'un bit de stuffing avant l'octet de CRC.

| donnée | 0x55 | 0x55 | 0xAA | 0xF3 | 0x59(CRC) |
|---|---|---|---|---|---|
| transmission | 10101010 | 10101010 | 01010101 | 11001111(0) | 10011010 |
| ***Line*** | 01100110 | 01100110 | 11001100 | 001000001 | 10111001 |


4) <u>stuffing au milieu de l'octet de CRC</u>

Je transmets les données permettant l'apparition d'un bit de stuffing au milieu de l'octet de CRC.

| donnée | 0xFC | 0x01 | 0x00 | 0x00 | 0xFD |
|---|---|---|---|---|---|
| transmission | 001111(0)11 | 10000000 | 00000000 | 00000000 | 101111(0)11 |
| ***Line*** | 100000111 | 10101010 | 10101010 | 10101010 | 011111000 |


5) <u>stuffing a la fin de l'octet de CRC</u>

Comme le bit de stuffing est inséré avant le 5$^{ième}$ bit de données à '1', et que le fanion de start commence toujours par un 0, le cas d'un bit de stuffing en fin de CRC ne peut jamais arriver.

# V) Synthèse logique et Analyse temporelle

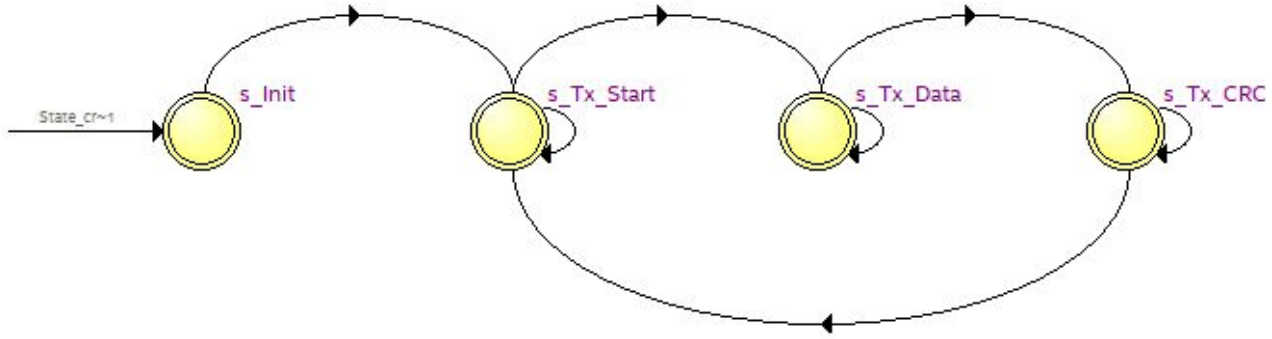
Maintenant que la simulation fonctionnelle a été validée, il faut vérifier que le code VHDL est compatible avec la phase de synthèse logique.

J'utilise donc le logiciel Quartus afin d'analyser et synthétiser le composant et pouvoir ainsi détecter des erreurs de conception et les corriger avant de continuer son développement. Suite à cette manipulation, je constate qu'aucune erreur ou "warning" n'est signalé par la compilation. Je sais dorénavant que le circuit est synthétisable.

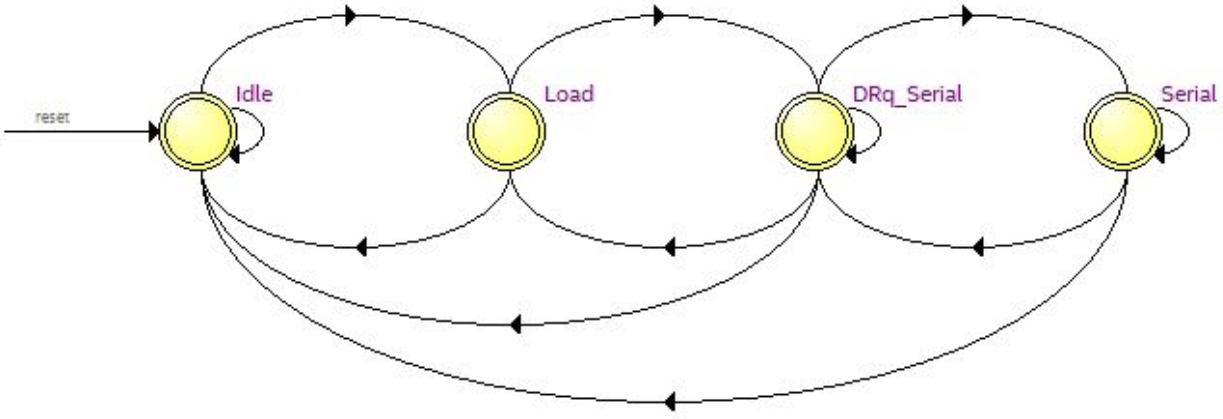
Ceci permet également de vérifier par exemple si les machines à états ont correctement été codées.

J'affiche donc les schémas synthétisés par le logiciel :

- Machine à état ELN :

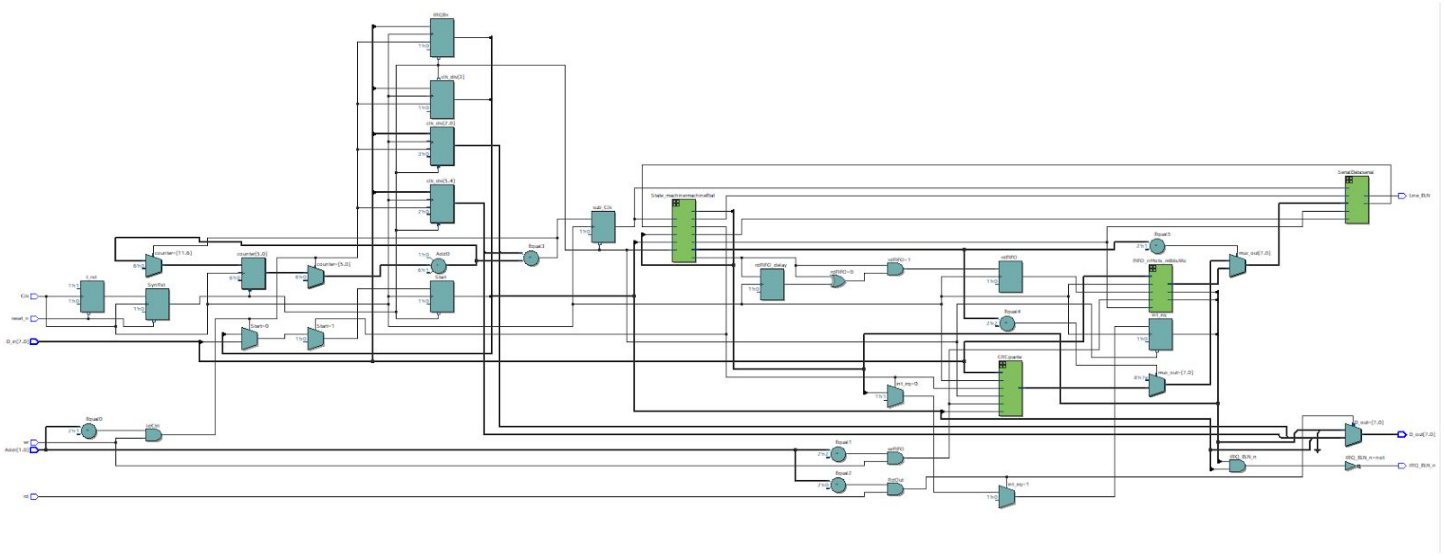



- Machine à états Serial :



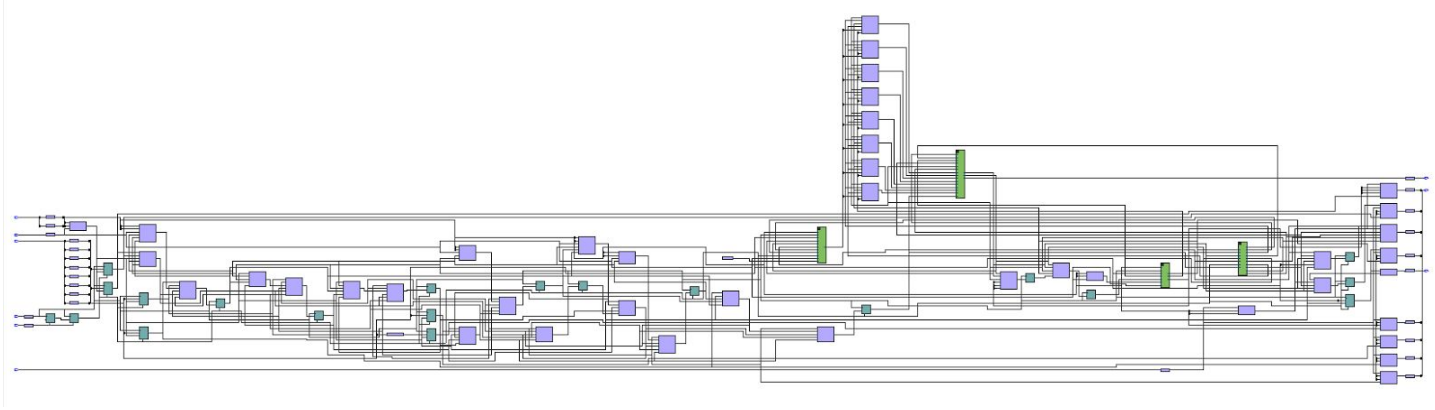

Dans les 2 cas, je retombe sur mes schémas présentés précédemment, ce qui valide mon implémentation. Les transitions revenant vers l'état **Idle** correspondent au signal *Init* qui force un retour à l'état initial.

Je regarde également le schéma logique synthétisé en utilisant la visualisation RTL.

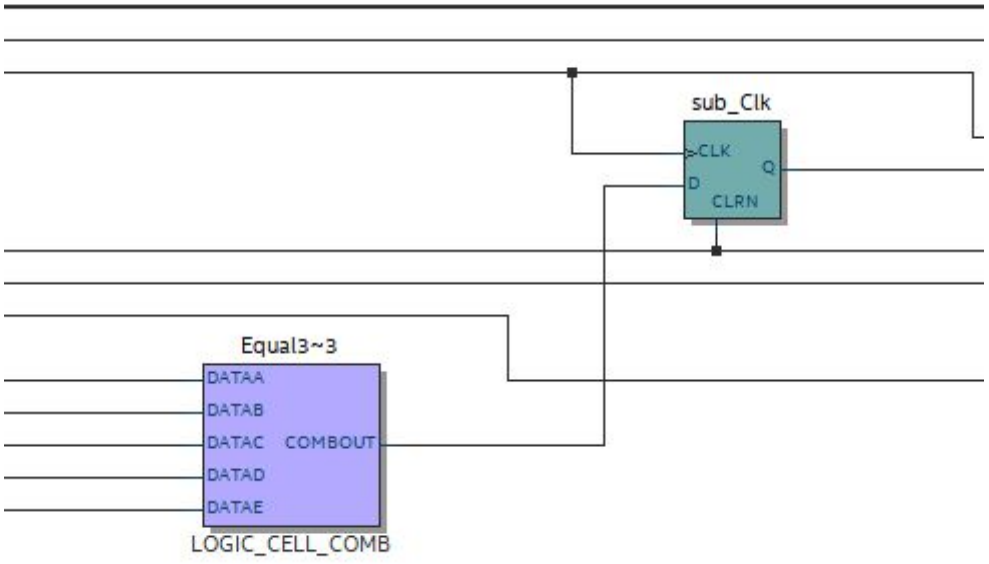


Même si l'agencement est différent, je reconnais très bien le schéma complet avec chaques registres et entités.

Un visualisation technologique du mapping permet de vérifier plus en détails que, par exemple, un registre a bien été utilisé en sortie du bloc combinatoire du diviseur d'horloge.



En regardant *sub_clk* :



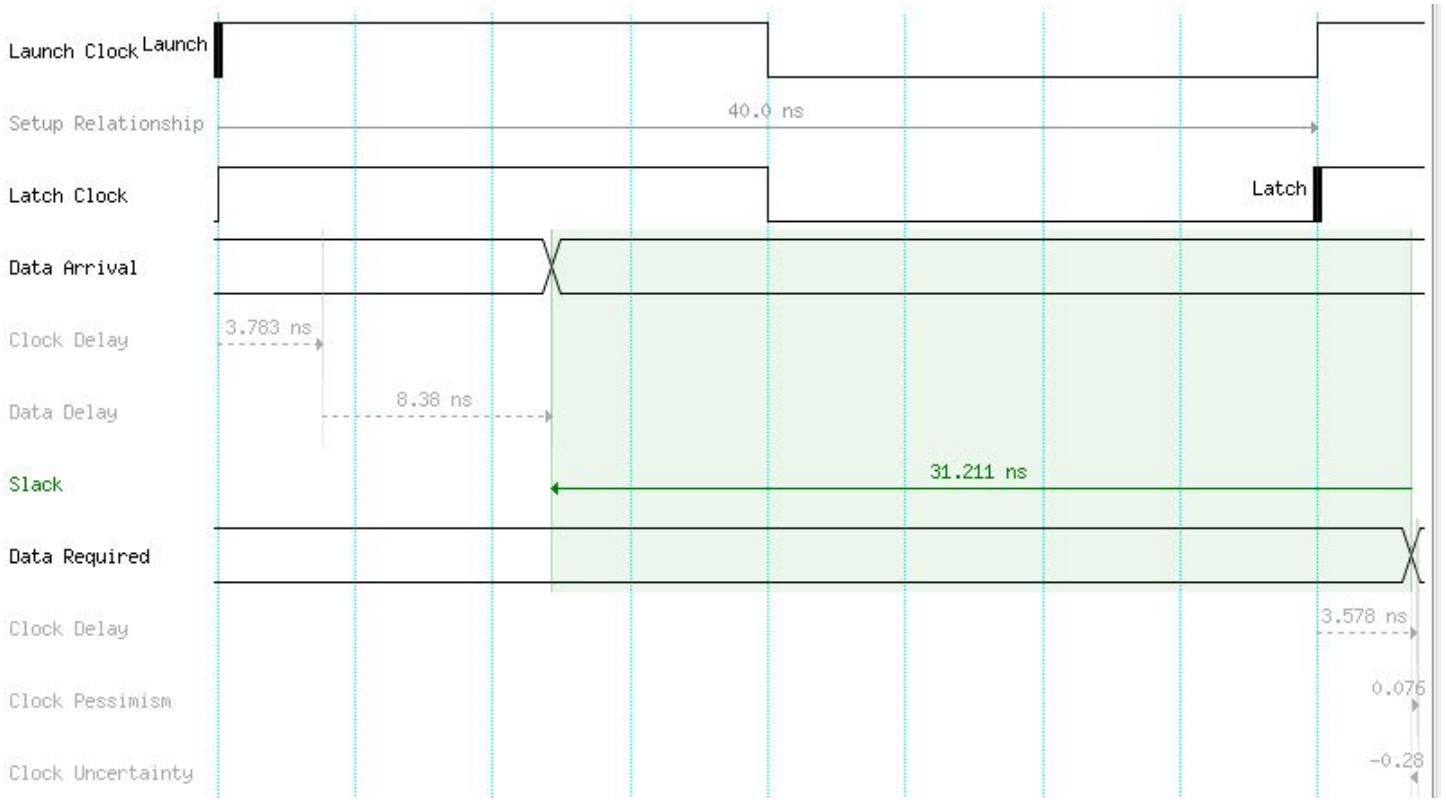
Il faut ensuite vérifier que les caractéristiques temporelles du schéma logique généré sont compatibles avec le cahier des charges. Je crée donc un fichier contenant les contraintes temporelles du système. Avec un processeur cadencé à 25MHz et le cas le plus contraignant du *clk_div* = 2, je définie les périodes d'horloge de clock à 40ns et de sub_clk à 80ns avec une "clock uncertaincy".
Il est très important de déclarer les 2 horloges pour que la synthèse puisse se faire correctement.

Le résultat de l'analyse temporelle avec l'outils TimeQuest montre qu'il y a aucun problème quant aux placements et maintiens des données.

| Operating Corner | Clock setup slack (ns) | Clock hold slack (ns) | Sub_clk setup slack (ns) | Sub_clk hold slack (ns) |
|---|---|---|---|---|
| Slow 85°C | 31.211 | 0.263 | 31.860 | 0.719 |
| Slow 0°C | 31.476 | 0.247 | 31.732 | 0.702 |
| Fast 85°C | 34.374 | 0.143 | 35.529 | 0.255 |
| Fast 0°C | 35.237 | 0.079 | 35.782 | 0.229 |

Toutes les marges sont positives quelque soit les conditions de fonctionnement choisies.

Je peux même si je le souhaite visualiser par exemple une représentation graphique du "setup slack" en slow corner à 85°C pour être sûr de la signification de la valeur.

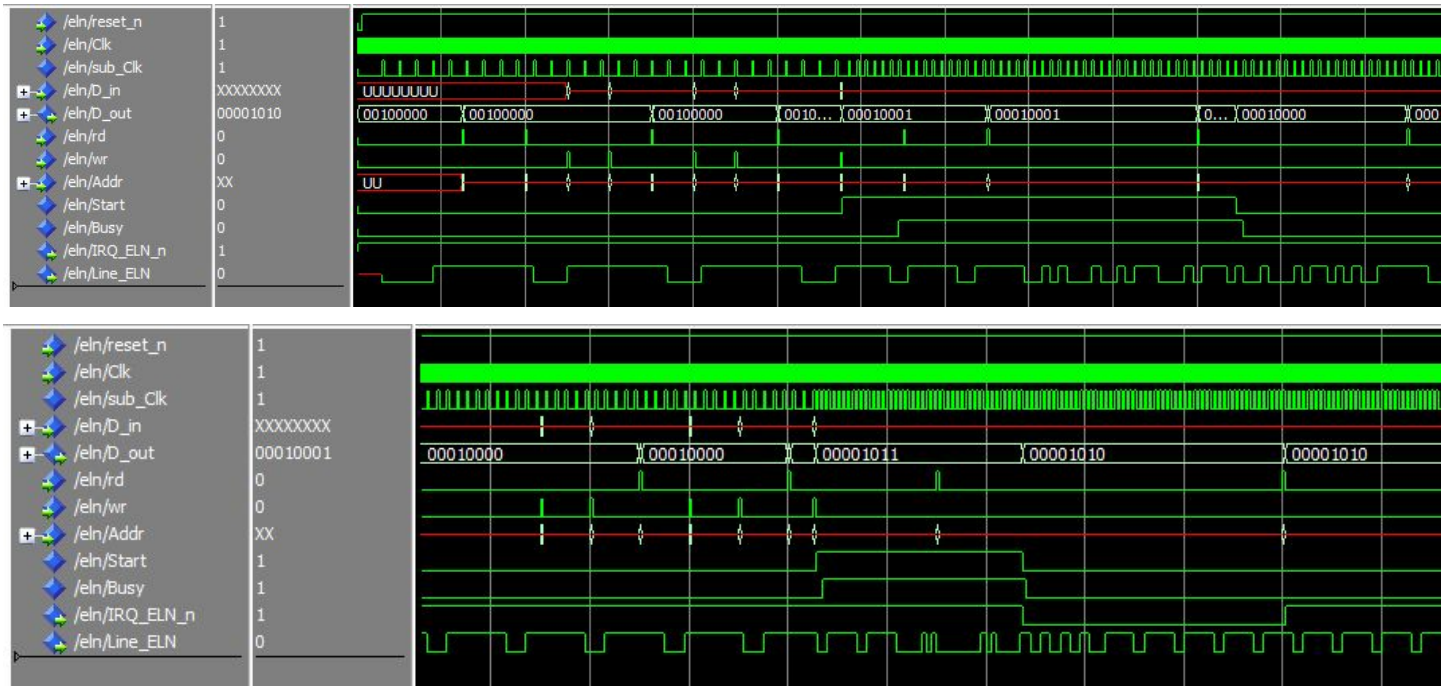


# VI) Simulation structurelle

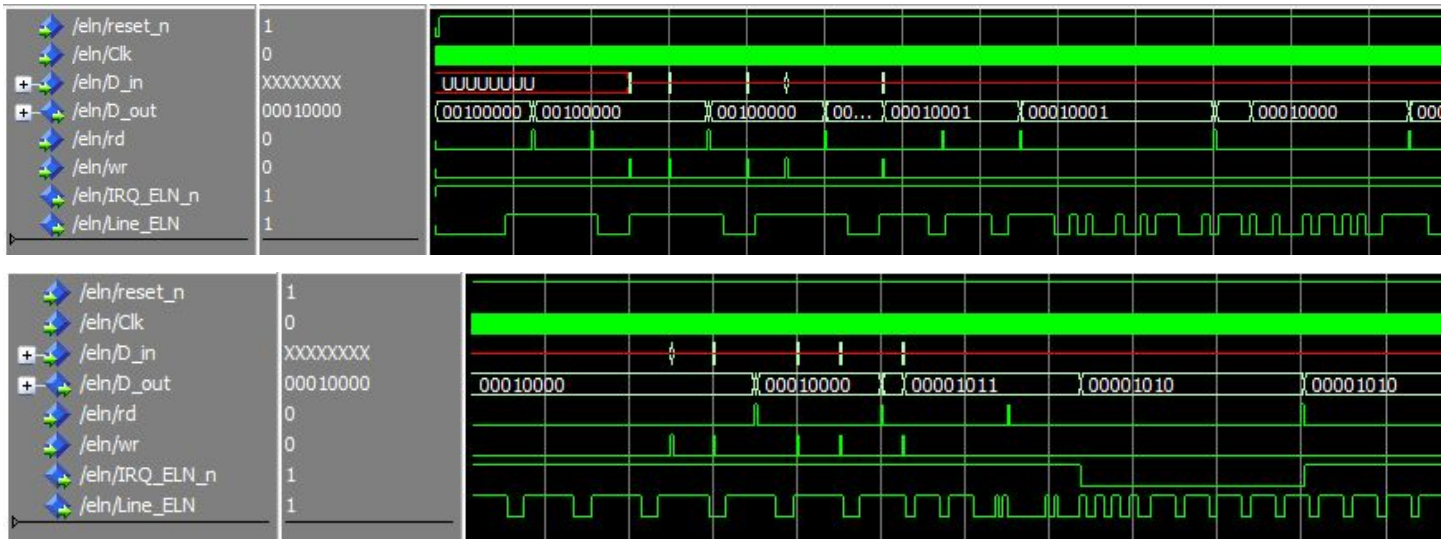
Maintenant que le fonctionnement du composant semble correcte en tous points, je décide de re-simuler un test à partir du fichier *.vho* généré par la compilation du projet composant par Quartus. Seul le test global sera re-réalisé car il permet de faire une vérification d'ensemble et si il est validé, alors tous les sous-composants le sont, en partant du principe que la synthèse du composant a donc été réalisée correctement à partir des codes.

Pour ce faire, j'utilise un scénario quelconque, avec des niveaux de sortie ***Line*** comportants des bits de stuffing et un changement d'horloge. Je teste ce scénario également avec la simulation fonctionnelle qui sert d'étalon comme elle a déjà été validée et avec laquelle je pourrai comparer le résultat.

Résultat fonctionnel




Résultat structurel

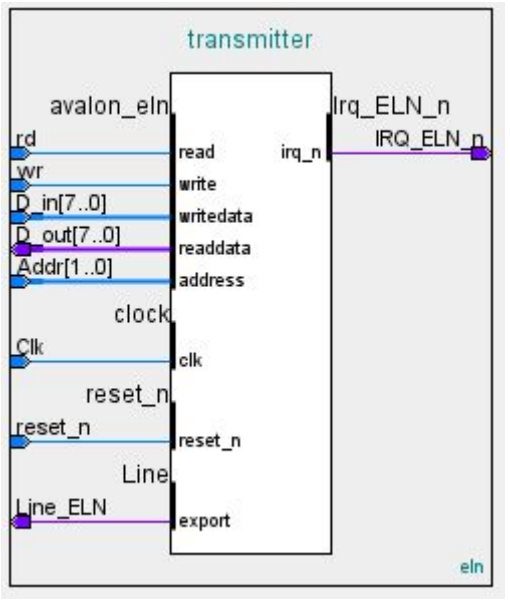



J'observe des chronogrammes parfaitement similaires. J'en conclue que la synthèse du composant a été réalisée avec succès. Il peut maintenant être instancié dans un système.

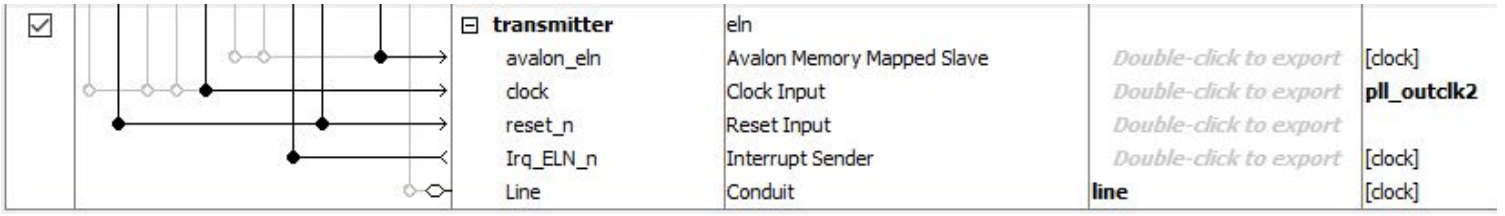
# VII) Création et Instanciation du composant

Après avoir créé le composant sous Platform Designer (anciennement Qsys) conformément au cahier des charges, je peux l'instancier dans le système global qui est la carte FPGA de développement Cyclone V.

La schématisation du composant créé dans le logiciel est la suivante:




Cela correspond bien à la représentation du composant qui avait été donnée. Je le connecte donc au système avec un niveau d'IRQ à 3:




Il faut maintenant connecter les signaux dans l'entité de plus haut niveau ainsi que le port GPIO externe qui portera le signal ***Line***.

Déclaration du nouveau signal:

```
-- serial
signal eln_line: std_logic;
```

Dans le composant DE0_CV_QSYS :

```
line_export : out   std_logic;      -- export
line_export    => eln_line
```

Connection du signal

```
-- Serial GPIO
GPIO_1(35) <= eln_line;
```

Le signal de sortie ***Line*** de l'émetteur pour liaison numérique existe dorénavant dans l'entité de plus haut niveau et est connecté à un port physique.

La création de ce composant a également généré une publication des propriétés du composant : `eln_hw.tcl` résumant toute la configuration effectuée.

# VIII) Test de l'interface Avalon

Une fois que le composant a été entièrement instancié, il est utile de vérifier si tous ses ports ont été correctement connectés. Pour cela, j'effectue un test de l'interface Avalon avec mon code téléchargé sur la carte de développement.

Comme pour les simulations des chronogrammes, j'utilise une macro de test qui sera écrite en TCL:

- Lecture status
- Lecture control
- Écriture de 2 données 0x61 0x62 ('a' & 'b')
- Lecture status
- Écriture de 2 données 0x63 0x64 ('c' & 'd')
- Lecture status
- Écriture control 0x13 (clk_div = 4 | IrqEn = 1 | start = '1')
- Lecture status
- Lecture status
- Lecture control
- Lecture status

j'obtiens en réponse :

```
*** test: ELN ***
Status: 0x02                  ->  Empty | !Full | !busy | !IRQ
Control: 0x20                 -> clk_div = 8 | !IRQEn | !start (valeurs par défaut)
write 0x61
write 0x62
Status: 0x00                  -> !Empty | !Full | !busy | !IRQ
write 0x63
write 0x64
Status: 0x04                  -> !Empty |  Full | !busy | !IRQ
START write control : 0x13    -> clk_div = 4 |  IrqEn |  start
Status: 0x0a                  ->  Empty | !Full | !busy |  IRQ
Status: 0x02                  ->  Empty | !Full | !busy | !IRQ  (IRQ a bien été lu)
Control: 0x12                 -> clk_div = 4 |  IrqEn | !start = 0

Master service closed
```

Les réponses reçues démontrent que le composant et les registres réagissent correctement.
Je suis désormais certain du bon fonctionnement de l'interconnexion "System designer ↔Avalon".

# IX) Définition des fichiers des macros et des APIs

Pour que l'ELN puisse être utilisé au niveau logiciel, il faut d'abord développer les propriétés de son pilote et son API. Les codes complets sont joints en annexe III.

## a) Propriétés

Pour pouvoir être intégré comme partie du Board Support Package (BSP), il faut créer son *driver*. Il faut donc déclarer certaines propriétés tel que son nom, le nom du composant, sa version, ou encore les fichiers qui implémentent ses APIs. Ce fichier de publication des propriétés du pilote, nommé `eln_sw.tcl` est joint en annexe III.

## b) APIs

Pour commencer, il faut regrouper dans un fichier ELN_regs.h tous les numéros des registres du composant ainsi que les masques et offsets des données contenues.

Par exemple, pour le registre ***ELNControl*** :

```
#define     ELNControl                          1

//ELNControl Register, 1, R/W
#define     IORD_ELN_CONTROL(base)              IORD_8DIRECT(base, ELNControl)
#define     IOWR_ELN_CONTROL(base, reg)         IOWR_8DIRECT(base, ELNControl, reg)

//CTRL Register, WO/RO
#define     ELN_CTRL_START_MSK                  (0x01)
#define     ELN_CTRL_START_OFST                 (0)
#define     ELN_CTRL_IRQEN_MSK                  (0x02)
#define     ELN_CTRL_IRQEN_OFST                 (1)
#define     ELN_CTRL_CLK_DIV_MSK                (0xFC)
#define     ELN_CTRL_CLK_DIV_OFST               (2)
```

De cette manière, le code des APIs se servira de ces macros et si une des valeurs doit changer, il ne faudra la modifier uniquement dans ce fichier.

Je peux maintenant déclarer toutes les fonctions nécessaires à la bonne et complète manipulation du composant. Par soucis de facilité d'utilisation, j'ai fait le choix d'implémenter un accesseur et un mutateur pour chaque paramètre. L'utilisateur pourra donc se servir du groupe de fonction suivant :

```
////////////////////////////////////////////////////////////
/// INIT FUNCTION
////////////////////////////////////////////////////////////
char ELN_Init(unsigned long, long, long, alt_isr_func);

////////////////////////////////////////////////////////////
/// STATUS REGISTER
////////////////////////////////////////////////////////////
alt_u8 ELN_Status(unsigned long);
char ELN_isBusy(unsigned long);
char ELN_isEmpty(unsigned long);
char ELN_isFull(unsigned long);
char ELN_IRQState(unsigned long);
```

```
//////////////////////////////////////////////////////////////
/// CONTROL REGISTER
//////////////////////////////////////////////////////////////
char ELN_setControl(unsigned long, alt_u8 );
alt_u8 ELN_getControl(unsigned long);
char ELN_setClock_div(unsigned long, alt_u8 );
alt_u8 ELN_getClock_div(unsigned long);
char ELN_setIrqEn(unsigned long,char);
char ELN_getIrqEn(unsigned long);
char ELN_setStart(unsigned long, char);
char ELN_getStart(unsigned long);

//////////////////////////////////////////////////////////////
/// D_IN REGISTER
//////////////////////////////////////////////////////////////
alt_u8 ELN_writeData(unsigned long, unsigned alt_u8 );
```

Il faut également déclarer une macro qui créera l'instance du composant et une autre qui permettra d'installer une routine d'interruption.

```
#define ELN_INSTANCE_INIT(name)                         \
      ELN_Init(                                         \
            name##_BASE,                                \
            name##_IRQ_INTERRUPT_CONTROLLER_ID,         \
            name##_IRQ,                                 \
            name##_ISR                                  \
      );


#define ELN_ISR_INSTALL(name, cb)                       \
      void name##_ISR(void* context){                   \
            IORD_ELN_STATUS(name##_BASE);               \
            cb();                                       \
    }
```

Le composant est désormais totalement fonctionnel, autant d'un point de vue matériel que logiciel. Un code pourra être développé et exécuté avec la carte FPGA pour sérialiser et émettre des données choisies par l'utilisateur.

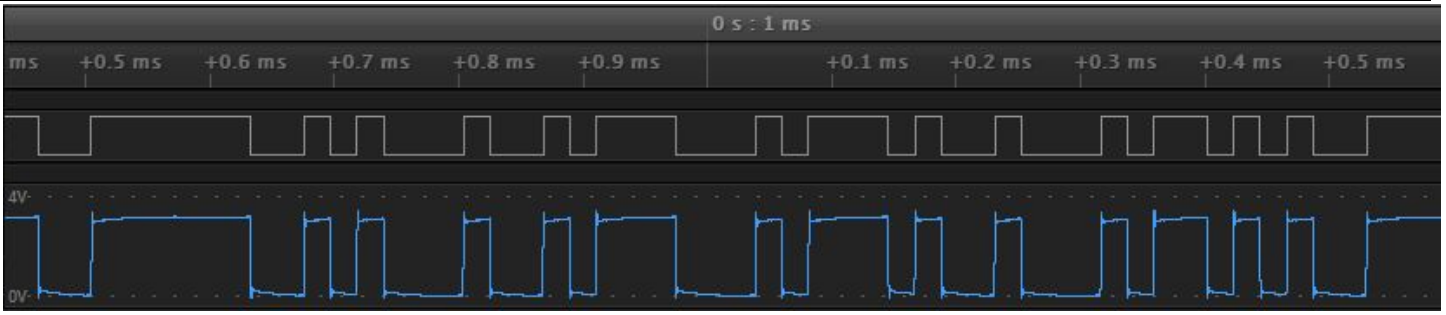
# X) Visualisations des données par analyseur logique

Même si le composant est enfin entièrement prêt à l'utilisation, j'effectue cependant encore quelques tests afin de vérifier le bon fonctionnement final de l'émetteur à liaison numérique. Afin de vérifier toute la chaîne qui a été développée, j'implémente un code faisant appel aux APIs et je regarde en sortie les signaux physique présents sur la pin GPIO grâce à une analyseur logique Saleae. Afin de pouvoir les observer, j'émets les message en boucle.

## a) Transmission lente

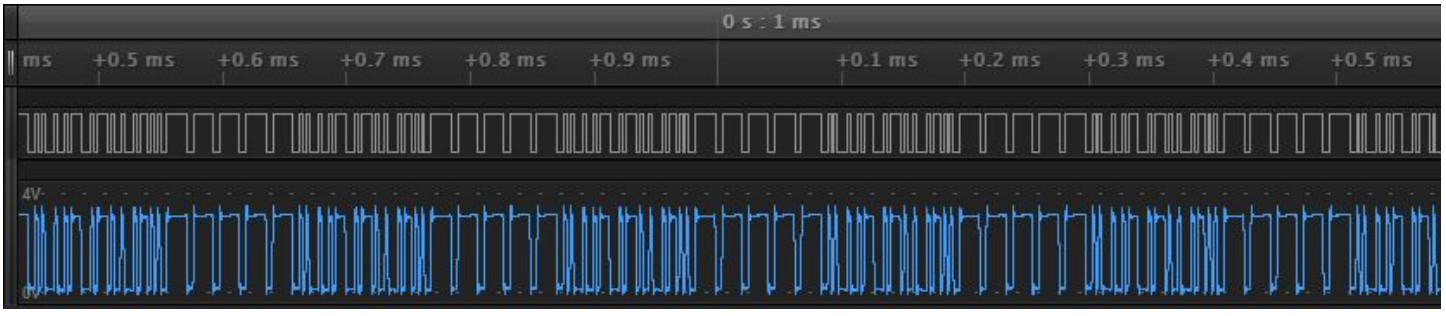
Tout d'abord, je transmets 4 caractères "abcd" à faible débit (*clk_div* = 32) sur ***Line*** afin de pouvoir bien visualiser les niveaux. Je rappelle que les niveaux attendus sont :

| Fanion | 1er octet | 2eme octet | 3eme octet | 4eme octet | CRC |
|---|---|---|---|---|---|
| 01111110 | 01010001 | 00101110 | 00101110 | 10010001 | 01101010 |




## b) Transmission rapide

J'essaie la même transmission à un cadencement plus élevé (*clk_div* = 4) afin d'observer l'influence sur le signal de sortie.

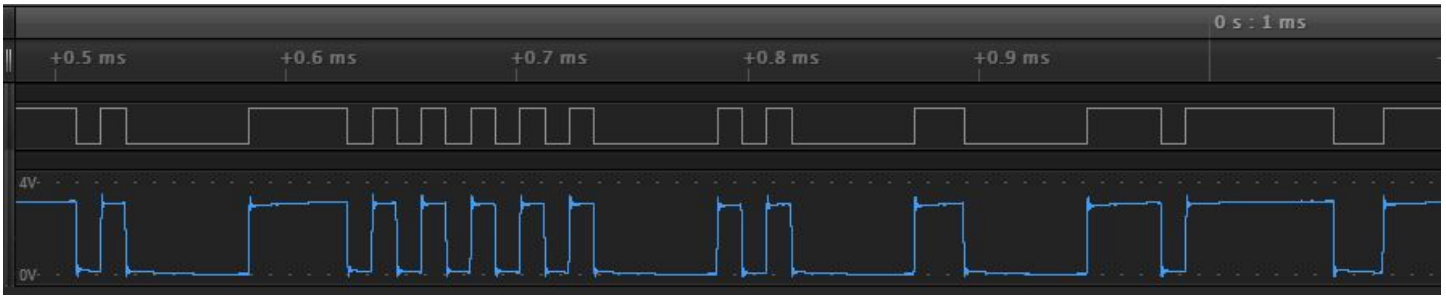



On peut observer sur le signal analogique que la qualité du signal et des fronts est dégradée à cause que l'augmentation de la fréquence d'émission.

## c) Transmission avec des bits de stuffing

Pour aussi observer l'apparition de bits de stuffing sur la trame, j'ai choisi un signal à envoyer qui présente toutes les situations de stuffing possible, même si cela ne correspond pas à une chaîne de caractère standard.

| donnée | 0xFC | 0x01 | 0xF0 | 0xF0 | 0xFD |
|---|---|---|---|---|---|
| transmission | 001111(0)11 | 10000000 | 00001111 | 00001111(0) | 101111(0)11 |
| ***Line*** | 100000111 | 10101010 | 10100000 | 101000001 | 100000111 |



# XI) Programme utilisateur

Le développement du composant est maintenant entièrement finalisé et validé. Pour finir, il est intéressant d'implémenter un petit code permettant à l'utilisateur d'envoyer du texte sur l'Émetteur à Liaison Numérique depuis un terminal. Quand l'utilisateur appuiera sur entrée (*carriage return : '\r'* ), les données seront envoyés. Je me place dans le cas où les caractères saisis seront transférés dans la pile dans la limite de sa capacité puis émis sur ***Line***. Lorsqu'ils auront tous été transmis, l'utilisateur pourra saisir une nouvelle chaîne. Le code `main.c` est joint en annexe IV.

Ce code ne transmettant que les données une seule fois, j'utilise un analyseur logique embarqué, déclenchant la capture sur le signal start afin de voir facilement la trame.

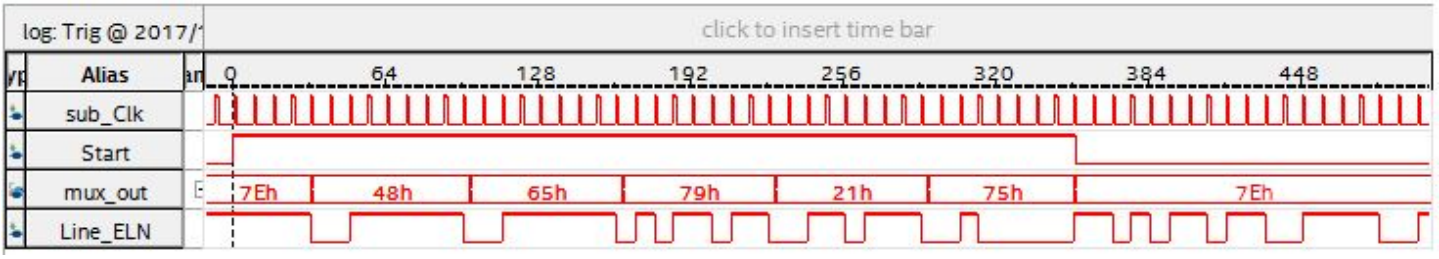
Je décide d'envoyer le message "Hey!" (0x48 0x65 0x79 0x21) avec le terminal.

```
*** START OF PROGRAM ***
Initialisation...
setting clock divisor...
Enabling IRQ...

enter string to send :
sending 4 bytes
Hey!
Data transmitted succesfully

-----------------------

enter string to send :
```

Les niveaux à observer avec l'analyseur logique embarqué sont :

| donnée | 0x48 | 0x65 | 0x79 | 0x21 | 0x75 |
|---|---|---|---|---|---|
| transmission | 00010010 | 10100110 | 10011110 | 10000100 | 10101110 |
| ***Line*** | 10110110 | 01101110 | 01000001 | 10101101 | 10011110 |

Ce sont bien les niveaux présent sur ***Line***.



Je décide ensuite de tester ensuite l'envoi d'un chaine plus longue et non multiple de la capacité de la pile : "Bonjour".

```
enter string to send :
sending 7 bytes
Bonj
Data transmitted succesfully
our
Data transmitted succesfully

----------------------

enter string to send :
```

En envoyant la chaine "Bonjour", deux envois distincts sont visibles, le deuxième n'ayant que 3 caractères.

Du côté de l'analyseur, les deux trames sont bien reconnaissables:


Après plusieurs essais infructueux, les signaux envoyés sont trop difficilement visualisables par l'analyseur logique physique Saleae pour je puisse en joindre une image. Ceci est dû au fait que l'analyseur possède une mémoire limitée qui empêche de le laisser enregistrer suffisamment longtemps pour que je puisse lancer et capturer un envoi unique des données.

# XII) Synthèse et Conclusion

Pour synthétiser ce rapport, j'ai donc développé un composant physique sur un circuit de développement FPGA, partant de la conception du schéma de l'architecture jusqu'au code d'interface logiciel (sur terminal) afin que ce composant soit parfaitement fonctionnel et utilisable. J'ai dû effectuer toutes les étapes de développement et tests afin de m'assurer qu'il réponde en tous points au cahier des charges qui m'a été imposé.

Je peux conclure ce compte-rendu en apportant un regard critique sur ce projet.

Premièrement, même si cet exercice s'est révélé plein de tournants inattendus ( difficulté de choix de solutions, prise en charge du projet en autonomie totale, changements soudain d'interprétation du cahier des charges…. ), il fut très formateur. En effet, il m'a tout d'abord permis de pouvoir mener un projet intégralement, des prémices de l'étude du système jusqu'aux mesures réelles sur des signaux physiques. Ceci donne une impression d'aboutissement, de pouvoir avoir littéralement entre les mains une réalisation de plusieurs semaines de travail et de s'en servir. Les attentes que j'en avais sont pleinement satisfaites et plus encore. Il m'a également poussé à affronter un certain challenge et m'a appris qu'il y a toujours une solution, il faut juste parfois penser à regarder différemment. J'ai également essayé de ne pas tomber dans la facilité et de ne pas me limiter au sujet en ne développant qu'un composant hyper-spécifique et non ré-utilisable.

Cependant, je suis conscient que ma solution proposée n'est peut-être pas idéale. Malgré le fait que je me sois efforcé de développer de manière souple et pourtant robuste, il y a certaines améliorations que j'aurais pu apporter. Par exemple, une implémentation plus réactive pourrait être proposée entre la demande d'un envoi d'une trame et la transmission réelle des données, ma solution devant laisser le contenu des deux registres présents dans l'entité serial être envoyés avant de sérialiser la donnée. Un moyen de *flusher* ces registres pour charger directement la donnée par accès rapide pourrait être intéressant. Cette dernière proposition devant en revanche gérer ce processus de "court-circuit" entre la fin d'un fanion de start et avant la prochain sérialisation….

Il y également le polynôme pour le calcul du CRC qui va à l'encontre de la généricité de mon code. Étant pour l'instant codé en valeur fixe dans le bloc de calcul, un quatrième registre sur l'interface Avalon permettrait de modifier le polynôme, ou au moins avoir le choix dans un nombre restreint de polynômes connus. Ainsi le composant serait totalement flexible au choix de l'utilisateur, ce qui apporte un vrai plus au projet.

Merci de m'avoir suivi sur ce travail.

Richard Taupiac

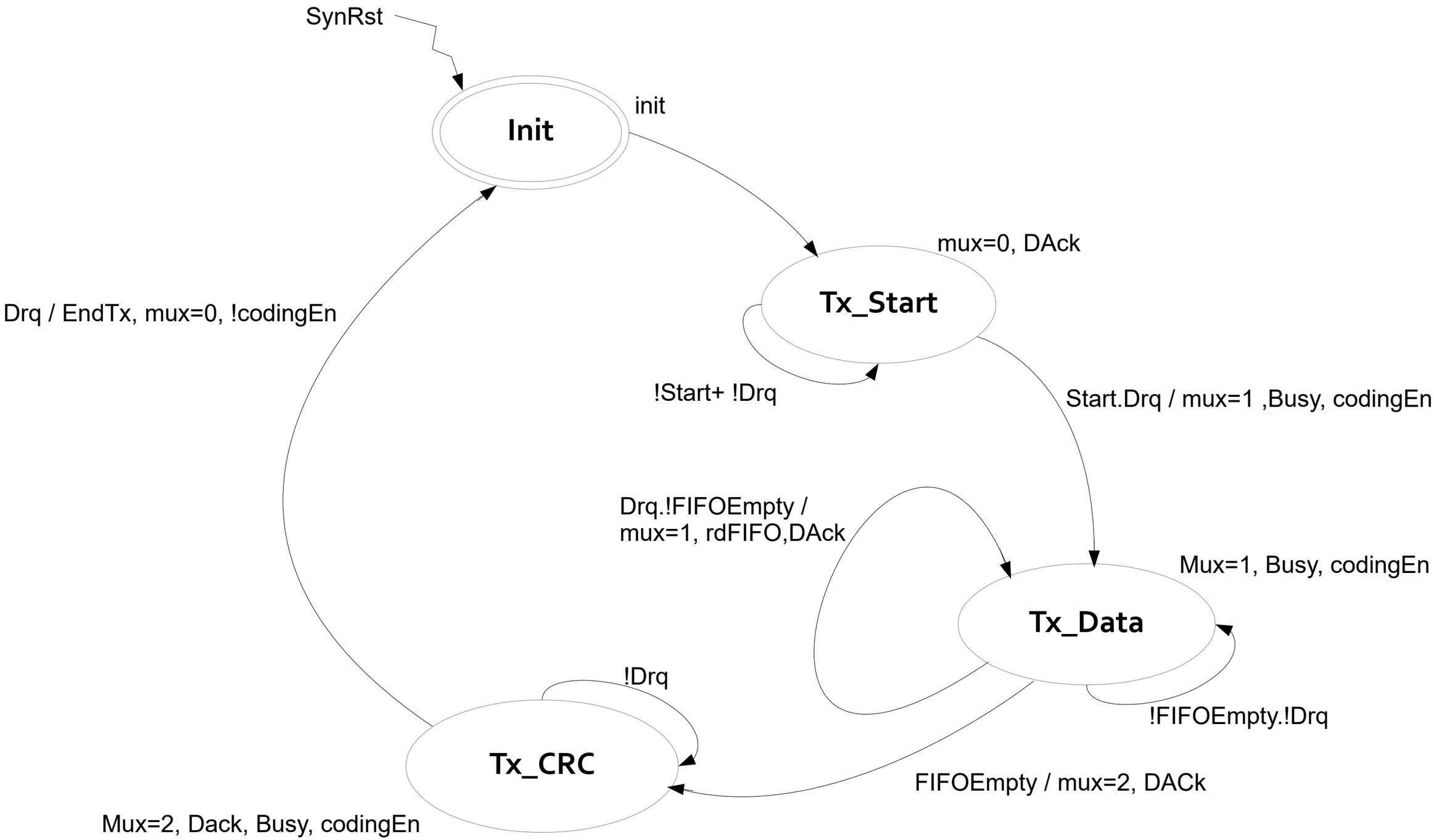
# XIII) Annexes

I. Schémas (5 pages)
   A. Graphe des états général ELN
   B. Schéma général ELN - 1
   C. Schéma général ELN - 2
   D. Graphe des états Serial - UC
   E. Schéma Serial - UT

II. Code source VHDL (9 pages)
   A. ELN.vhd
   B. SerialData.vhd
   C. FIFO_nMots_mBits.vhd

III. Code source logiciel (5 pages)
   A. eln.sw.tcl
   B. ELN_regs.h
   C. ELN_API.c
   D. ELN_API.h

IV. Code utilisateur (1 page)
   A. main.c



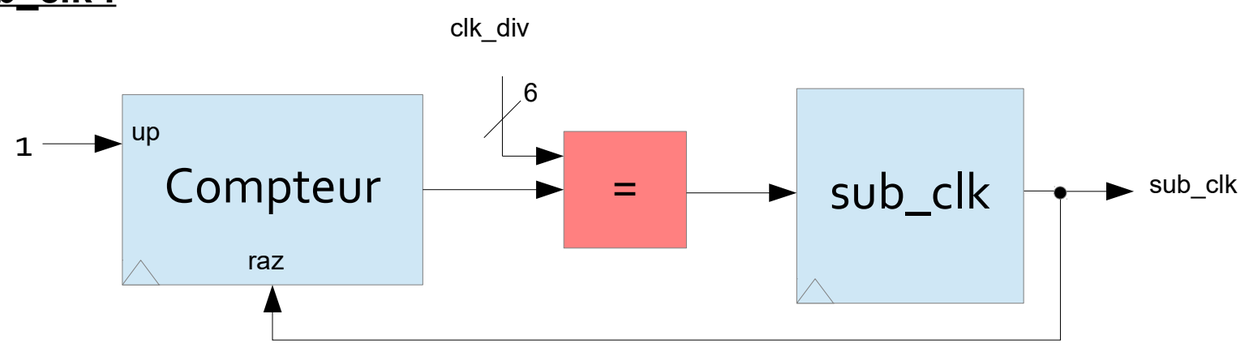

# Graphe des états général ELN

Légende :
clock synchrone
sub_clk synchrone
Combinatoire

D_in
8
D_out
8
1
0
8
8
RdOut
[0]
[1]
[2]
[3]
[7..4]
« 0000 »
IRQF
raz
RdOut
SynRst
EndTx
reset
raz
Start
[0]
[0]
IdCtrl
SynRst
reset
IrqEn
[1]
[1]
IdCtrl
6
6
SynRst
reset
clk_div
[7..2]
[7..2]
IdCtrl
Irq_ELN_n
FIFOEmpty
FIFOFull
Din
FIFO
Dout
init
wrFIFO
8
rdFIFO
Trigger
rdFIFO_sub
DataIn
CRC
Id
start
raz
wrFIFO
start
EndTx
8
10
01
00
mux_out
« 01111110 »
8
2
EndTx
Busy
Machine Etats
Reset
Start
SynRst
Start
mux
init
codingEn
DAck
DRq
Serial
Line

# Schéma général ELN - 1

## sub_clk :

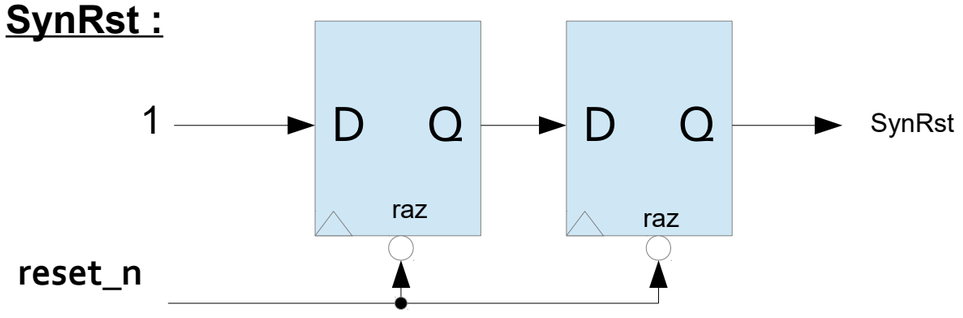



Légende :

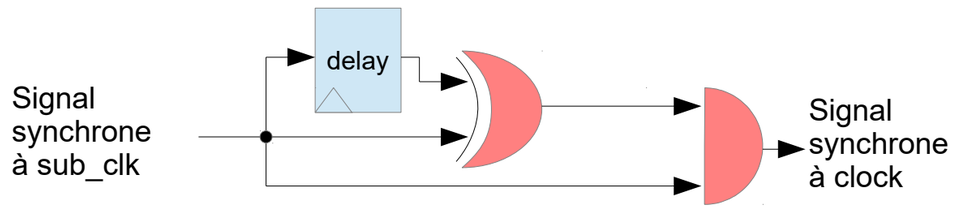
- clock synchrone
- sub_clk synchrone
- Combinatoire

## Decodeur :


## SynRst :



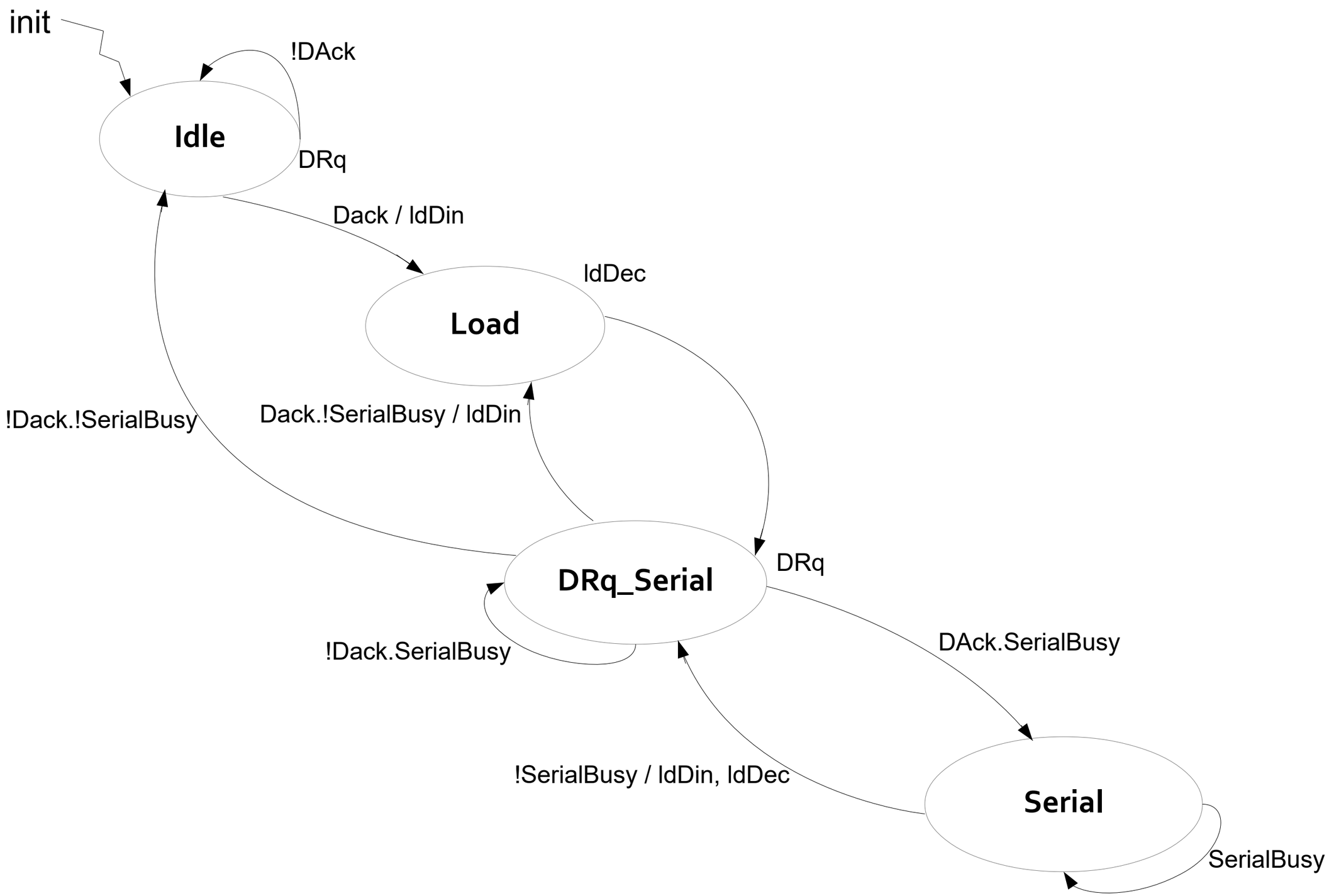

## Trigger :




## Equations de décodage :

- RdOut = 1 si Addr = 0 et rd = 1
  sinon 0
- ldCtrl = 1 si Addr = 1 et wr = 1
  sinon 0
- wrFIFO = 1 si Addr = 2 et wr = 1
  sinon 0

# Schéma général ELN - 2




# Graphes des états SERIAL - UC

Légende :

- serial_clk synchrone*
- Combinatoire

*(*) : considéré comme « serial_clock synchrone » : éléments travaillant avec la clock injecté dans l'unité SERIAL*

Din

8

codingEn

in

ld **codingIn**

ldDin

in

ld **Data**

in

ld **codingDec**

8

in

ldDec

ld **Dec** out

dec next

1

0

0

enable

in **codage** out

Dout

b2 b1 b0

raz

Compteur up

SerialBusy

<

8

# Schéma SERIAL - UT

```vhdl
-------------------------------------------------------------------------------
-- ELN
-- Project : ELN
-- Date : 22/12/2017
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
-- Machine A Etats
-------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity State_machine is
    port(   Clk, SynRst, Start, DRq, FIFOEmpty : in std_logic;
            init, DAck, rdFIFO, EndTx, Busy, codingEn : out std_logic;
            mux : out std_logic_vector(1 downto 0)
    );
end entity State_machine;

architecture rtl of State_machine is
    type States is (s_Init, s_Tx_Start, s_Tx_Data, s_Tx_CRC);
    signal State_cr, State_sv : States;
begin
    RegStates : process(Clk, SynRst)

    begin
        if SynRst = '0' then
                State_cr <= s_Init;
        elsif rising_edge(Clk) then
            State_cr <= State_sv;
        end if;
    end process RegStates;

    process(State_cr, Start, DRq, FIFOEmpty)
    begin
        init <= '0'; DAck <= '0'; rdFIFO <= '0'; EndTx <= '0'; Busy <= '0'; codingEn
        <= '0'; mux <= "00";
        State_sv <= State_cr;
        case State_cr is
            when s_Init =>  init <= '1'; State_sv <= s_Tx_Start;

            when s_Tx_Start =>
                    mux <= "00"; DAck <= '1';
                    if Start = '1' and DRq = '1' then
                        State_sv <= s_Tx_Data;
                        Busy <= '1';
                        mux <= "01";
                        codingEn <= '1';
                    end if;

            when s_Tx_Data =>
                    mux <= "01"; Busy <= '1'; codingEn <= '1';
                    if FIFOEmpty = '1' then
                        mux <= "10"; DAck <= '1';
                        State_sv <= s_Tx_CRC;
                    elsif DRq = '1' then
                        mux <= "01"; DAck <= '1';
                        rdFIFO <= '1';
                    end if;

            when s_Tx_CRC =>
                    mux <= "10"; DAck <= '1'; Busy <= '1'; codingEn <= '1';
                    if DRq = '1' then
                        State_sv <= s_Tx_Start;
                        EndTx <= '1'; mux <= "00";
                        codingEn <= '0';
                    end if;
        end case;

    end process;
end architecture rtl;

```

```
73  ------------------------------------------------------------------------------
74  -- CRC
75  ------------------------------------------------------------------------------
76  library IEEE;
77  use IEEE.std_logic_1164.all;
78  use IEEE.numeric_std.all;
79
80  entity CRC is
81      generic (DATA_WIDTH : integer := 8;
82               SIZE : integer := 4);
83      port(   Clk, SynRst, RAZ, ldCRC, start : in std_logic;
84          CRC_in : in std_logic_vector(DATA_WIDTH-1 downto 0);
85          CRC_out : out std_logic_vector(DATA_WIDTH-1 downto 0)
86      );
87  end entity CRC;
88
89  architecture rtl of CRC is
90      signal CRC_reg : std_logic_vector((SIZE*DATA_WIDTH)+7 downto 0);
91      signal poly : std_logic_vector(DATA_WIDTH downto 0) := "100000001";
92
93  begin
94      process(Clk, SynRst)
95      variable level : integer range 0 to (SIZE*DATA_WIDTH);
96      begin
97
98          if SynRst = '0' then
99              CRC_reg <= (others=>'0');
100             level := (SIZE*DATA_WIDTH);
101
102         else
103             if rising_edge(Clk) then
104                 if RAZ = '1' then
105                     CRC_reg <= (others=>'0');
106                     level := SIZE*DATA_WIDTH;
107
108                 elsif ldCRC = '1' and level > 7 then
109                     CRC_reg(level+7 downto level) <= CRC_in;
110                     CRC_reg(level-1 downto 0) <= (others=>'0');
111                     level := level - 8;
112
113                 elsif start = '1' and level < (SIZE*DATA_WIDTH) then
114                     if(CRC_reg((SIZE*DATA_WIDTH)+7) = '1') then -- if MSB = 1
115                         CRC_reg((SIZE*DATA_WIDTH)+7 downto 0) <=
                            (CRC_reg((SIZE*DATA_WIDTH)+6 downto ((SIZE-1)*DATA_WIDTH)+7)
116                                                 xor poly(DATA_WIDTH-1 downto
                                                    0))&CRC_reg(((SIZE-1)*DATA_WIDTH)+6
                                                    downto 0)&'0';
117                         level := level + 1;
118
119                     else
120                         -- shift
121                         CRC_reg((SIZE*DATA_WIDTH)+7 downto 0) <=
                            CRC_reg((SIZE*DATA_WIDTH)+6 downto 0)&'0';
122                         level := level + 1;
123                     end if;
124
125                 end if;
126             end if;
127         end if;
128     end process;
129
130     CRC_out <=  CRC_reg((SIZE*DATA_WIDTH)+7 downto ((SIZE-1)*DATA_WIDTH)+8);
131
132 end architecture rtl;
133
134 ------------------------------------------------------------------------------
135 -- ELN
136 ------------------------------------------------------------------------------
137
138 library IEEE;
139 use IEEE.std_logic_1164.all;
140 use IEEE.numeric_std.all;
141 use work.all;
```

```
142
143  entity eln is
144          port(   Clk, reset_n, rd, wr : in std_logic;
145          Line_ELN, IRQ_ELN_n : out std_logic;
146          D_in : in std_logic_vector(7 downto 0);
147          D_out : out std_logic_vector(7 downto 0);
148          Addr : in std_logic_vector(1 downto 0)
149          );
150  end entity eln;
151
152  architecture rtl of eln is
153          signal rdFIFO, wrFIFO, FIFOFull, FIFOEmpty, init, EndTx, DAck, DRq, t_rst,
             SynRst, Start,ldCtrl, RdOut,
154          Busy, int_irq, IRQEn, codingEn, sub_Clk, rdFIFO_sub, rdFIFO_delay : std_logic;
155      signal clk_div : std_logic_vector(5 downto 0) := "001000";
156      signal mux : std_logic_vector(1 downto 0);
157      signal FIFOout : std_logic_vector(7 downto 0);
158      signal mux_out, CRC_out : std_logic_vector(7 downto 0);
159
160  begin
161      serial :    entity SerialData
162                  generic map (   DATA_WIDTH => 8)
163                  port map(   Clk => sub_Clk,
164                          Init => init,
165                          DAck => DAck,
166                          codingEn => codingEn,
167                          DIn => mux_out,
168                          DOut => Line_ELN,
169                          DRq => DRq
170                          );
171
172      machineEtat :   entity State_machine
173                  port map(   Clk => sub_Clk,
174                          SynRst => SynRst,
175                          init => init,
176                          Start => Start,
177                          DRq => DRq,
178                          FIFOEmpty => FIFOEmpty,
179                          DAck => DAck,
180                          rdFIFO => rdFIFO_sub,
181                          EndTx => EndTx,
182                          Busy => Busy,
183                          codingEn => codingEn,
184                          mux => mux
185                          );
186
187      parite : entity CRC
188                  generic map (   DATA_WIDTH => 8,
189                          SIZE => 4
190                          )
191                  port map(   Clk => Clk,
192                          SynRst => SynRst,
193                          RAZ => endTx,
194                          start => Start,
195                          ldCRC => wrFIFO,
196                          CRC_in =>  D_in,
197                          CRC_out => CRC_out
198                      );
199
200      fifo :  entity FIFO_nMots_mBits
201                  generic map (   DATA_WIDTH => 8,
202                          FIFO_SIZE => 2
203                          )
204                  port map(   Horloge => Clk,
205                          initFifo => init,
206                          WrFifo => wrFIFO,
207                          RdFifo => rdFIFO,
208                          DataIn => D_in,
209                          DataOut => FIFOout,
210                          FifoEmpty => FIFOEmpty,
211                          FifoFull => FIFOFull
212                          );
213
```

```
214   -- signal sub_clock interface
215
216       process(Clk)
217       begin
218           if rising_edge(Clk) then
219                   rdFIFO_delay <= rdFIFO_sub;
220                   rdFIFO <= (rdFIFO_sub xor rdFIFO_delay) and rdFIFO_sub;
221               end if;
222       end process;
223
224
225   -- Decodeur
226
227       ldCtrl <= '1' when Addr = "01" and Wr='1'
228       else '0';
229
230       wrFIFO <= '1' when Addr = "10" and Wr='1'
231       else '0';
232
233       RdOut <= '1' when Addr = "00" and Rd='1'
234       else '0';
235
236
237   -- Start
238
239       process(Clk, SynRst)
240       begin
241           if SynRst = '0' then
242               start <= '0';
243           else
244
245               if rising_edge(Clk) then
246                   if EndTx = '1' then
247                       start <= '0';
248                   elsif ldCtrl = '1' then
249                       start <= D_in(0);
250                   end if;
251               end if;
252
253           end if;
254       end process;
255
256
257   -- Clock Divisor
258
259       process(Clk, SynRst)
260       variable counter : integer range 0 to 63;
261       begin
262           if SynRst = '0' then
263               clk_div <= "001000";
264               counter := 0;
265               sub_Clk <='0';
266
267           elsif rising_edge(Clk) then
268               if ldCtrl = '1' then
269                   clk_div <= D_in(7 downto 2);
270               end if;
271
272               counter := counter + 1;
273
274               if counter >= to_integer(unsigned(clk_div)) then
275                   sub_Clk <='1';
276                   counter := 0;
277               else
278                   sub_Clk <='0';
279               end if;
280
281           end if;
282
283       end process;
284
285
286
```

```
287   -- IRQEn
288
289       process(Clk, SynRst)
290       begin
291           if SynRst = '0' then
292               IRQEn <= '0';
293           else
294               if rising_edge(Clk) then
295                   if ldCtrl = '1' then
296                       IRQEn <= D_in(1);
297                   end if;
298               end if;
299           end if;
300       end process;
301
302
303   -- IRQF
304
305       process(Clk, SynRst)
306       begin
307           if SynRst = '0' then
308               int_irq <= '0';
309           else
310               if rising_edge(Clk) then
311                   if RdOut = '1' then
312                       int_irq <= '0';
313                   elsif EndTx = '1' then
314                       int_irq <= '1';
315                   end if;
316               end if;
317           end if;
318       end process;
319
320
321   -- IRQ
322       IRQ_ELN_n <= not(int_irq and IRQEn);
323
324
325   -- MUX
326       D_out <=    clk_div&IRQEn&start when RdOut = '0' else
327               "0000"&int_irq&FIFOFull&FIFOEmpty&Busy;
328
329       mux_out <=  FIFOout when mux = "01" else
330               CRC_out when mux = "10" else
331               "01111110";
332
333
334   -- "Synchronous deactivation" Reset
335
336       process(Clk, reset_n)
337       begin
338           if reset_n = '0' then
339               t_rst <= '0'; SynRst <= '0';
340           elsif rising_edge(Clk) then
341               t_rst <= '1';
342               SynRst <= t_rst;
343           end if;
344       end process;
345
346
347   end architecture rtl;
348
349
```

```
1  ----------------------------------------------------------------------------------
2  -- SERIALDATA
3  -- Project : ELN
4  -- Date : 22/12/2017
5  ----------------------------------------------------------------------------------
6
7  ----------------------------------------------------------------------------------
8  -- UC
9  ----------------------------------------------------------------------------------
10
11 library IEEE;
12 use IEEE.std_logic_1164.all;
13 use IEEE.numeric_std.all;
14
15 entity UC_SerialData is
16         port(   Clk, Init, DAck, serialBusy : in std_logic;
17         DRq, LdDIn, LdDec : out std_logic
18             );
19 end entity UC_SerialData;
20
21 architecture rtl of UC_SerialData is
22     type Etats is (Idle, Load, DRq_Serial, Serial);
23     signal Etat_cr, Etat_sv : Etats;
24 begin
25     RegEtat : process(Clk)
26     begin
27         if rising_edge(Clk) then
28             if Init = '1' then
29                 Etat_cr <= Idle;
30             else
31                 Etat_cr <= Etat_sv;
32             end if;
33         end if;
34     end process RegEtat;
35
36     process(Etat_cr, DAck, serialBusy)
37     variable state : std_logic_vector(1 downto 0);
38     begin
39         DRq <= '0'; LdDIn <= '0'; LdDec <= '0';
40         Etat_sv <= Etat_cr;
41         state := DAck & serialBusy;
42         case Etat_cr is
43             when Idle =>        DRq <= '1';
44                         if DAck = '1' then
45                             LdDIn <= '1';
46                             Etat_sv <= Load;
47                         end if;
48
49             when Load =>        LdDec <= '1';
50                         Etat_sv <= DRq_Serial;
51
52             when DRq_Serial =>  DRq <= '1';
53                         case state is
54                             when "10" =>    LdDIn <= '1'; Etat_sv <= Load;
55                             when "11" =>    Etat_sv <=
                               Serial;
56                             when "00" =>    Etat_sv <= Idle;
57                             when others =>  NULL;
58                         end case;
59
60             when Serial =>      if serialBusy = '0' then
61                             ldDec<= '1'; LdDIn <= '1';
62                             Etat_sv <= DRq_Serial;
63                         end if;
64         end case;
65     end process;
66 end architecture rtl;
67
68
69
70
71
72
```

```
73  ----------------------------------------------------------------------------------
74  -- UT
75  ----------------------------------------------------------------------------------
76
77  library IEEE;
78  use IEEE.std_logic_1164.all;
79  use IEEE.numeric_std.all;
80
81  entity UT_SerialData is
82      generic (DATA_WIDTH : integer := 8);
83      port(   Clk, LdDIn, LdDec, codingEn : in std_logic;
84              DIn : in std_logic_vector(DATA_WIDTH-1 downto 0);
85              DOut, serialBusy : out std_logic
86      );
87  end entity UT_SerialData;
88
89  architecture rtl of UT_SerialData is
90      signal Data, Dec : std_logic_vector(DATA_WIDTH-1 downto 0) := (others=>'0');
91      signal out_b, State, codingIn, codingDec : std_logic ;
92      signal Delay_b0, Delay_b1, Delay_b2, next_b : std_logic;
93
94  begin
95      Registres : process(Clk)
96      variable stuffing : std_logic;
97      variable counter : integer range 0 to DATA_WIDTH;
98      begin
99          if rising_edge(Clk) then
100
101
102             stuffing := Delay_b2 and Delay_b1 and Delay_b0 and out_b and next_b;
103
104             -- update delays registers
105             Delay_b2 <= Delay_b1;
106             Delay_b1 <= Delay_b0;
107             Delay_b0 <= out_b;
108
109             if LdDIn = '1' then -- load Din
110                 Data <= DIn;
111                 codingIn <= codingEn;
112             end if;
113
114             if LdDec = '1' then -- load Dec and first shift
115                 counter := 0;
116                 codingDec <= codingIn;
117
118                 if stuffing = '1' and codingIn = '1' then -- if five '1' were
                    transmitted while coding enabled
119                     Dec(DATA_WIDTH-1 downto 0) <= Data(DATA_WIDTH-1 downto 0);
120                     out_b <= '0'; -- write 0
121                     stuffing := '0';
122                 else -- shift
123                     Dec(DATA_WIDTH-2 downto 0) <= Data(DATA_WIDTH-1 downto 1);
124                     Dec(DATA_WIDTH-1) <= '0';
125                     counter := counter + 1;
126                     out_b <= Data(0);
127                     next_b <= Data(1);
128                 end if;
129             else
130
131                 if stuffing = '1' and codingDec ='1' then -- if five '1' were
                    transmitted while coding enabled
132                     out_b <= '0'; -- write 0
133                     stuffing := '0';
134                 else -- shift
135                     Dec(DATA_WIDTH-2 downto 0) <= Dec(DATA_WIDTH-1 downto 1);
136                     Dec(DATA_WIDTH-1) <= '0';
137                     counter := counter + 1;
138                     out_b <= Dec(0);
139
140                     if counter < DATA_WIDTH then next_b <= Dec(1);
141                     else next_b <= Data(0); end if;
142                 end if;
143             end if;
```

```
144
145         if counter < DATA_WIDTH then serialBusy <= '1'; else serialBusy <='0'; end if;
146
147         if (codingDec ='1' or (LdDec = '1' and codingIn = '1')) then
148             State <= not(out_b xor State) ;
149         else
150             State <= out_b;
151         end if;
152
153
154         end if;
155     end process Registres;
156
157     -- Line out
158     Dout <= State;
159
160 end architecture rtl;
161
162 ---------------------------------------------------------------------------------
163 -- Global
164 ---------------------------------------------------------------------------------
165
166 library IEEE;
167 use IEEE.std_logic_1164.all;
168 use IEEE.numeric_std.all;
169 use work.all;
170
171 entity SerialData is
172     generic (DATA_WIDTH : integer := 8);
173         port(   Clk, Init, DAck, codingEn : in std_logic;
174                 DIn : in std_logic_vector(DATA_WIDTH-1 downto 0);
175                 DOut, DRq : out std_logic
176         );
177 end entity SerialData;
178
179 architecture rtl of SerialData is
180     signal LdDIn,LdDec,serialBusy : std_logic;
181
182 begin
183     UT :    entity UT_SerialData
184             generic map (   DATA_WIDTH => DATA_WIDTH)
185             port map(   Clk => Clk,
186                         DIn => DIn,
187                         DOut => DOut,
188                         LdDIn => LdDIn,
189                         LdDec => LdDec,
190                         codingEn => codingEn,
191                         serialBusy => serialBusy
192                     );
193
194     UC :    entity UC_SerialData
195             port map(   Clk => Clk,
196                         Init => Init,
197                         DRq => DRq,
198                         DAck => DAck,
199                         LdDIn => LdDIn,
200                         LdDec => LdDec,
201                         serialBusy => serialBusy
202                     );
203 end architecture rtl;
```

```vhdl
--------------------------------------------------------------------------------
-- FIFO GENERIC
-- Project : ELN
-- Date : 22/12/2017
--------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity FIFO_nMots_mBits is
  generic (DATA_WIDTH : integer := 8;
           FIFO_SIZE : integer := 3);
    port(Horloge, initFifo, WrFifo, RdFifo : in std_logic;
         DataIn : in std_logic_vector(DATA_WIDTH-1 downto 0);
         DataOut : out std_logic_vector(DATA_WIDTH-1 downto 0);
         FifoLevel : out std_logic_vector(FIFO_SIZE downto 0);
         FifoEmpty, FifoFull : out std_logic);
end entity FIFO_nMots_mBits;

architecture rtl of FIFO_nMots_mBits is
    signal adrFifo : integer range 0 to 2**FIFO_SIZE; --nombre de mots + position
    débordement
    type memory is array(0 to 2**FIFO_SIZE-1) of std_logic_vector(DATA_WIDTH-1
    downto 0);
    signal Fifo : memory;
begin
MyFifo : process(Horloge)
variable FifoAccess : std_logic_vector(1 downto 0);
begin
    if (rising_edge(Horloge)) then
      if initFifo = '1' then     --raz du compteur d'adresse
        adrFifo <= 0;
      else
        FifoAccess := WrFifo&RdFifo;
        case FifoAccess is
          when "10" =>                      --ECRITURE SEULE:
            if adrFifo < 2**FIFO_SIZE then
              Fifo(adrFifo) <= DataIn;       --FIFO <- DataIn
              adrFifo <= adrFifo+1;         --compteur d'adresse incrémenté
            end if;
          when "01" =>                      --LECTURE SEULE:
            if adrFifo > 0 then
              for i in 1 to 2**FIFO_SIZE-1 loop
                 Fifo(i-1) <= Fifo(i);        --décalage
              end loop;
              adrFifo <= adrFifo-1;         --compteur d'adresse décrémenté
            end if;
          when "11" =>                      --ECRITURE & LECTURE:
            for i in 1 to 2**FIFO_SIZE-1 loop
              Fifo(i-1) <= Fifo(i);         --décalage
            end loop;
            Fifo(adrFifo-1) <= DataIn;      --FIFO <- DataIn, décalage, adr ne change
            pas
          when others => NULL;
        end case;
      end if;
    end if;
end process MyFifo;

FifoEmpty <=  '1' when adrFifo = 0 else       --pile vide = adr à 0
              '0';

FifoFull <= '1' when adrFifo = 2**FIFO_SIZE else --pile pleine = toutes les adresses
utilisées
            '0';

DataOut <= Fifo(0);   --port de sortie = bas de la pile

FifoLevel <= std_logic_vector(TO_UNSIGNED(adrFifo, FIFO_SIZE+1));

end architecture rtl;
```

```
1  ##################################################
2  # ELN_driver.tcl                                 #
3  ##################################################
4
5  # Create a new driver
6  create_driver ELN_driver
7
8  # Associate it with some hardware known as "ELN"
9  set_sw_property hw_class_name ELN
10
11 # The version of this driver
12 set_sw_property version 17.0
13 set_sw_property min_compatible_hw_version 1.0
14
15 # Location in generated BSP that above sources will be copied into
16 set_sw_property bsp_subdirectory drivers
17
18 #this driver uses new API
19 set_sw_property supported_interrupt_apis enhanced_interrupt_api
20
21 # This driver supports HAL BSP (OS) type
22 add_sw_property supported_bsp_type HAL
23
24
25 ##################################################
26 # Source file listings...                        #
27 ##################################################
28
29 # Include files (*.h)
30 add_sw_property include_source inc/ELN_regs.h
31 add_sw_property include_source inc/ELN_API.h
32
33 # C/C++ source files
34 add_sw_property c_source src/ELN_API.c
35
36 # asm source files
37 #add_sw_property asm_source src/< asm file name >.s
38
39
40 # End of file
41
```

```
/****************************************************
**  ELN                                            **
**  8 bits device for Altera/Avalon interface      **
****************************************************

****************************************************
**  Register mapping                               **
****************************************************
*   Register      |  Reg Num  |  Access  |  Reset  *
*****************|***********|**********|**********
*   ELNStatus     |    0      |  R       | *
*   ELNControl    |    1      |  R/W     | *
*   ELNFIFO       |    2      |  W       | *
****************************************************/
//note 1: internal Ctrl register is initialized at 0


#ifndef __ELN_REGS_H__
#define __ELN_REGS_H__

#include <io.h>

#define ELNStatus       0
#define ELNControl      1
#define ELNFIFO         2


/****************************************************
**        Access Macros                            **
****************************************************/
//ELNStatus Register, 0, RO
#define IORD_ELN_STATUS(base)           IORD_8DIRECT(base, ELNStatus)

//ELNControl Register, 1, R/W
#define IORD_ELN_CONTROL(base)          IORD_8DIRECT(base, ELNControl)
#define IOWR_ELN_CONTROL(base, reg)     IOWR_8DIRECT(base, ELNControl, reg)

//ELNFIFO Register, 1, W0
#define IOWR_ELN_DATA(base, data)       IOWR_8DIRECT(base, ELNFIFO, data)

/****************************************************
**        Mask & Offsets                           **
****************************************************/
//ELNStatus Register, RO
#define ELN_STATUS_BUSY_MSK             (0x01)
#define ELN_STATUS_BUSY_OFST            (0)
#define ELN_STATUS_EMPTY_MSK            (0x02)
#define ELN_STATUS_EMPTY_OFST           (1)
#define ELN_STATUS_FULL_MSK             (0x04)
#define ELN_STATUS_FULL_OFST            (2)
#define ELN_STATUS_IRQ_MSK              (0x08)
#define ELN_STATUS_IRQ_OFST             (3)

//CTRL Register, WO/RO
#define ELN_CTRL_START_MSK          (0x01)
#define ELN_CTRL_START_OFST         (0)
#define ELN_CTRL_IRQEN_MSK          (0x02)
#define ELN_CTRL_IRQEN_OFST         (1)
#define ELN_CTRL_CLK_DIV_MSK            (0xFC)
#define ELN_CTRL_CLK_DIV_OFST           (2)

//ELNFIFO Register, R/W
#define ELN_DATA_MSK                (0xFF)
#define ELN_DATA_OFST                   (0)

#endif /* __ELN_REGS_H__ */

```

```c
/************************************************************
**        ELN API SOURCE                                  **
************************************************************/
#include <stdio.h>
#include "ELN_API.h"

////////////////////////////////////////////////////////////
/// INIT FUNCTION
////////////////////////////////////////////////////////////

char ELN_Init(unsigned long BaseAddress, long ELN_ic_id, long ELN_irq, alt_isr_func isr)
{
    // init interrupt
    if(alt_ic_isr_register(ELN_ic_id, ELN_irq, isr, NULL, NULL))return(-1);

    //clears flag
    ELN_IRQState(BaseAddress);

    return((char)0);
}

////////////////////////////////////////////////////////////
/// STATUS REGISTER
////////////////////////////////////////////////////////////

alt_u8 ELN_Status(unsigned long BaseAddress)
{
    return((alt_u8)IORD_ELN_STATUS(BaseAddress));
}

char ELN_isBusy(unsigned long BaseAddress)
{
    return((char)((IORD_ELN_STATUS(BaseAddress) & ELN_STATUS_BUSY_MSK) >> ELN_STATUS_BUSY_OFST));
}

char ELN_isEmpty(unsigned long BaseAddress)
{
    return((char)((IORD_ELN_STATUS(BaseAddress) & ELN_STATUS_EMPTY_MSK) >> ELN_STATUS_EMPTY_OFST));
}

char ELN_isFull(unsigned long BaseAddress)
{
    return((char)((IORD_ELN_STATUS(BaseAddress) & ELN_STATUS_FULL_MSK) >> ELN_STATUS_FULL_OFST));
}

char ELN_IRQState(unsigned long BaseAddress)
{
    return((char)((IORD_ELN_STATUS(BaseAddress) & ELN_STATUS_IRQ_MSK) >> ELN_STATUS_IRQ_OFST));
}


////////////////////////////////////////////////////////////
/// CONTROL REGISTER
////////////////////////////////////////////////////////////

char ELN_setControl_div(unsigned long BaseAddress, alt_u8 reg)
{
    alt_u8 clock_div = (reg >> ELN_CTRL_CLK_DIV_OFST) & ELN_CTRL_CLK_DIV_MSK;

    if (clock_div <= 1 || clock_div >= 64) return((char)-1);
    else
    {
        IOWR_ELN_CONTROL(BaseAddress, reg);
        return((char)0);
    }
}


```

```
69  alt_u8 ELN_getControl(unsigned long BaseAddress)
70  {
71      alt_u8 reg = ((ELN_getClock_div(BaseAddress)<<
        ELN_CTRL_CLK_DIV_OFST)&ELN_CTRL_CLK_DIV_MSK ) | ((ELN_getIrqEn(BaseAddress) <<
        ELN_CTRL_IRQEN_OFST)&ELN_CTRL_IRQEN_MSK) | ((ELN_getStart(BaseAddress) <<
        ELN_CTRL_START_OFST)&ELN_CTRL_START_MSK);
72      return(reg);
73  }
74
75
76  // Clock_div
77  char ELN_setClock_div(unsigned long BaseAddress, alt_u8 clock_div)
78  {
79      if (clock_div <= 1 || clock_div >= 64) return((char)-1);
80      else
81      {
82          alt_u8 reg = ((clock_div << ELN_CTRL_CLK_DIV_OFST)&ELN_CTRL_CLK_DIV_MSK ) |
            ((ELN_getIrqEn(BaseAddress) << ELN_CTRL_IRQEN_OFST)&ELN_CTRL_IRQEN_MSK) |
            ((ELN_getStart(BaseAddress) << ELN_CTRL_START_OFST)&ELN_CTRL_START_MSK);
83
84          IOWR_ELN_CONTROL(BaseAddress, reg);
85          return((char)0);
86      }
87  }
88
89
90  alt_u8 ELN_getClock_div(unsigned long BaseAddress)
91  {
92      return((alt_u8)((IORD_ELN_CONTROL(BaseAddress)& ELN_CTRL_CLK_DIV_MSK) >>
        ELN_CTRL_CLK_DIV_OFST)) ;
93  }
94
95
96  // IRQEn
97  char ELN_setIrqEn(unsigned long BaseAddress, char IrqEn)
98  {
99      alt_u8 reg = ((ELN_getClock_div(BaseAddress) <<
        ELN_CTRL_CLK_DIV_OFST)&ELN_CTRL_CLK_DIV_MSK ) | ((IrqEn <<
        ELN_CTRL_IRQEN_OFST)&ELN_CTRL_IRQEN_MSK)  | ((ELN_getStart(BaseAddress) <<
        ELN_CTRL_START_OFST&ELN_CTRL_START_MSK) << ELN_CTRL_START_OFST);
100
101     IOWR_ELN_CONTROL(BaseAddress, reg);
102     return((char)0);
103 }
104
105
106 char ELN_getIrqEn(unsigned long BaseAddress)
107 {
108     return((char)((IORD_ELN_CONTROL(BaseAddress) & ELN_CTRL_IRQEN_MSK ) >>
        ELN_CTRL_IRQEN_OFST));
109 }
110
111
112 // START
113 char ELN_setStart(unsigned long BaseAddress, char Start)
114 {
115     alt_u8 reg = ((ELN_getClock_div(BaseAddress)<<
        ELN_CTRL_CLK_DIV_OFST)&ELN_CTRL_CLK_DIV_MSK ) | ((ELN_getIrqEn(BaseAddress) <<
        ELN_CTRL_IRQEN_OFST)&ELN_CTRL_IRQEN_MSK) | ((Start <<
        ELN_CTRL_START_OFST)&ELN_CTRL_START_MSK);
116
117     IOWR_ELN_CONTROL(BaseAddress, reg);
118     return((char)0);
119 }
120
121
122 char ELN_getStart(unsigned long BaseAddress)
123 {
124     return((char)((IORD_ELN_CONTROL(BaseAddress) & ELN_CTRL_START_MSK ) >>
        ELN_CTRL_START_OFST));
125 }
126
127
```

```
128
129 ////////////////////////////////////////////////////////////
130 /// DATAIN REGISTER
131 ////////////////////////////////////////////////////////////
132
133 char ELN_writeData(unsigned long BaseAddress, alt_u8 data)
134 {
135     // return error if FIFO is Full
136     if ( ELN_isFull(BaseAddress) ) return((char)-1);
137
138     // else write data
139     else IOWR_ELN_DATA(BaseAddress, data & ELN_DATA_MSK); return(0);
140 }
141
```

```
/*************************************************************
**        ELN API HEADER                                    **
*************************************************************/
#ifndef __ELN_API_H__
#define __ELN_API_H__

#include "ELN_regs.h"
#include "sys/alt_irq.h"        // interrupts

/*************************************************************
    ELN_ISR_INSTALL: this macro installs the ELN Interrupt Service Routine
        name: instance name used in QSys (UPPER CASE LETTERS) (it will be extended
        with _ISR)
        cb: callback function
*************************************************************/
#define ELN_ISR_INSTALL(name, cb)           \
    void name##_ISR(void* context){         \
        IORD_ELN_STATUS(name##_BASE);       \
        cb();                   \
    }

/*************************************************************
    ELN_INSTANCE_INIT: this macro initializes the ELN instance "name"
        name: instance name used in QSys (UPPER CASE LETTERS)
        Return: Success <=> 0, Error <=> -1
*************************************************************/
#define ELN_INSTANCE_INIT(name)             \
    ELN_Init(                   \
        name##_BASE,                \
        name##_IRQ_INTERRUPT_CONTROLLER_ID, \
        name##_IRQ,             \
        name##_ISR              \
    );


////////////////////////////////////////////////////////////
/// INIT FUNCTION
////////////////////////////////////////////////////////////
char ELN_Init(unsigned long, long, long, alt_isr_func);

////////////////////////////////////////////////////////////
/// STATUS REGISTER
////////////////////////////////////////////////////////////
alt_u8 ELN_Status(unsigned long);
char ELN_isBusy(unsigned long);
char ELN_isEmpty(unsigned long);
char ELN_isFull(unsigned long);
char ELN_IRQState(unsigned long);

////////////////////////////////////////////////////////////
/// CONTROL REGISTER
////////////////////////////////////////////////////////////
char ELN_setControl(unsigned long, alt_u8);
alt_u8 ELN_getControl(unsigned long);
char ELN_setClock_div(unsigned long, alt_u8);
alt_u8 ELN_getClock_div(unsigned long);
char ELN_setIrqEn(unsigned long,char);
char ELN_getIrqEn(unsigned long);
char ELN_setStart(unsigned long, char);
char ELN_getStart(unsigned long);

////////////////////////////////////////////////////////////
/// D_IN REGISTER
////////////////////////////////////////////////////////////
char ELN_writeData(unsigned long, alt_u8);

#endif
```

```c
// Global include
#include "DE0_CV_includes.h"
// API specific include
#include "ELN_API.h"

#define ms_sleep(delay)      (usleep(1000*delay))
#define sec_sleep(delay)     (ms_sleep(1000*delay))

void cb(void){
    printf("\nData transmitted succesfully\n");
    ELN_Status(TRANSMITTER_BASE); // clear flag
}

ELN_ISR_INSTALL(TRANSMITTER, cb);


int main(void){

    printf("\n*** START OF PROGRAM ***\n");

    printf("Initialisation...\n");
    ELN_INSTANCE_INIT(TRANSMITTER);  //Initialisation

    printf("setting clock divisor...\n");
    ELN_setClock_div(TRANSMITTER_BASE, 16); // set clock_div = 16

    printf("Enabling IRQ...\n");
    ELN_setIrqEn(TRANSMITTER_BASE, 1); // disable interruptions

    ms_sleep(500);

    int i;
    while(1)
    {
        //Ask for input string
        printf("\nenter string to send :\n");
        char* data;
        scanf("%s[^\r]", data);
        printf("sending %i bytes\n",strlen(data));

        for(i=0; i<strlen(data);)
        {

            //writing data if data is transmit, get next char
            if(ELN_writeData(TRANSMITTER_BASE,(alt_u8)data[i]) == 0)
            {printf("%c",(alt_u8)data[i]);i++;}


            //start transmitting if FIFO is full
            if(ELN_isFull(TRANSMITTER_BASE)) // if FIFO is full
            {
                ELN_setStart(TRANSMITTER_BASE,1);
                // wait for the end
                while(ELN_isBusy(TRANSMITTER_BASE)){}
            }

        }

        //transmit the rest if FIFO is not empty
        if(!ELN_isEmpty(TRANSMITTER_BASE)) // if FIFO is full
        {
            ELN_setStart(TRANSMITTER_BASE,1);
            // wait for the end
            while(ELN_isBusy(TRANSMITTER_BASE)){}
        }
        free(data);
        ms_sleep(50);
        printf("\n----------------------\n");
    }

    printf("\n*** END OF PROGRAM ***\n");
}
```