SEI-4101A TP CRC

September 21, 2023

DIDOUH Abel
abel.didouh@edu.esiee.fr

KHAZAAL Elio
elio.khazaal@edu.esiee.fr

Résumé

Dans les communications numériques, le Contrôle de Redondance Cyclique permet de détecter une erreur. Cependant cette méthode ne permet pas de corriger l'erreur. Pour mettre en œuvre le CRC-8CCITT nous allons réaliser l'architecture, la simulation et l'implémentation du code sur une carte de développement Basys3 construite autour d'un FPGA Xilinx Artix-7.

Architecture

1. On souhaite réaliser l'architecture du CRC-8CCITT dont le polynôme générateur est $g(x) = x^8 + x^2 + x + 1$. Le code RTL est dans le répertoire *rtl* et les simulations auront lieu dans le répertoire *tb* qui comprend les testbench et scripts TCL de simulation automatique. Dans le dossier *impl* nous aurons les fichiers d'implémentation.

   Nous souhaitons faire le calcul du reste de la division (sans gestion de la transmission série complète : envoi de la donnée en série puis du reste en série) ce qui implique qu'on a besoin d'utiliser le schéma de principe du CRC qui a pour équation: avec N = 8 dans notre cas (le degré du polynôme générateur). Cette architecture permettra de calculer le reste de la division de datain.g(x) par g(x). Puisque le message arrive en série sur 1 bit, ça veut dire qu'on a une donnée séquentielle en entrée. Donc, d'après l'équation qu'on a posée précédemment, on aura :

   $Q_7.g(x) = \text{data\_in}.(g(x) + x^8)$
   $Q_7.(x^8 + x^2 + x + 1) = \text{data\_in}.(x^8 + x^8 + x^2 + x + 1)$
   $Q_7.(x^8 + x^2 + x + 1) = \text{data\_in}.(2x^8 + x^2 + x + 1)$

   Puisque 2 modulo 2 est égal à 0, on obtient :
   $Q_7.(x^8 + x^2 + x + 1) = \text{data\_in}.(x^2 + x + 1)$

   En divisant par des deux côtés, on obtient :
   $Q_7.(1 + x^{-6} + x^{-7} + x^{-8}) = \text{data\_in}.(x^{-6} + x^{-7} + x^{-8})$

   On écrit en fonction de lui-même et de , on obtient :
   $Q_7 = x^{-8}.Q_7 + x^{-8}.\text{data\_in} + x^{-7}.Q_7 + x^{-7}.\text{data\_in} + x^{-6}.Q_7 + x^{-6}.\text{data\_in}$

   En appelant T la période de l'horloge clk, on écrit l'équation logique temporelle suivante :

   $$Q_7(t) = Q_7(t - 8T) \oplus \text{data\_in}(t - 8T) \oplus Q_7(t - 7T) \oplus \text{data\_in}(t - 7T) \oplus Q_7(t - 6T) \oplus \text{data\_in}(t - 6T)$$

   Comme l'architecture doit pouvoir calculer plusieurs restes successifs, elle nécessite une entrée supplémentaire init qui permet de réinitialiser le calcul. On a donc besoin d'ajouter un multiplexeur avec deux choix d'entrées avant chaque bascule contrôlée par init pour sélectionner si on veut réinitialiser toutes les bascules où sélectionner la sortie de la bascule précédente. On récupère le CRC des sorties Q0 à Q7 de chaque bascule. On obtient finalement le schéma suivant :

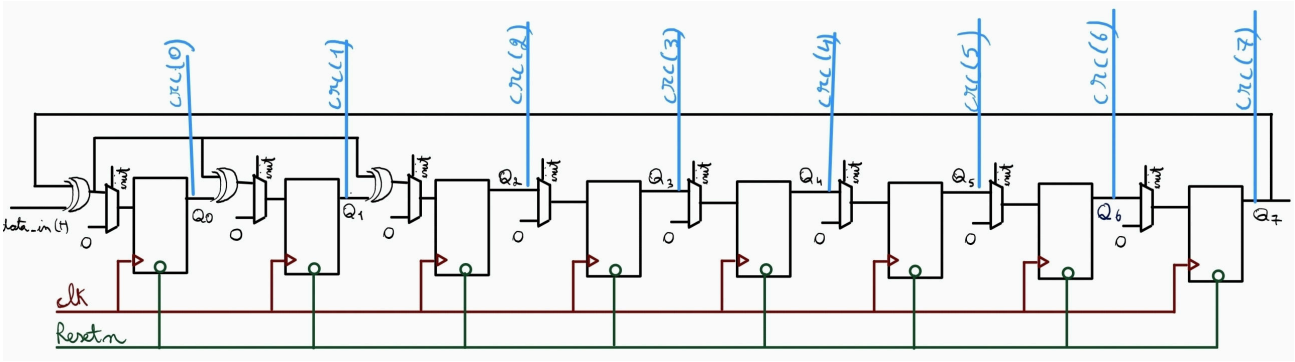

Figure 1: Schéma RTL du CRC-8CCITT

Simulation

1. Dans le testbench, il y a cinq messages de 40 bits. Nous allons calculer à la main le CRC des deux premiers vecteurs de test. Le premier vecteur en hexadécimal est le suivant : 0x0000000001. Pour obtenir le CRC à la main, nous devons diviser $x^{\deg(\text{polynome générateur})} \cdot \text{data\_in}$ par le polynome générateur. Pour le premier vecteur, nous avons le polynôme suivant : X^0 que l'on multiplie par X^8 et on divise ensuite par le polynome générateur.

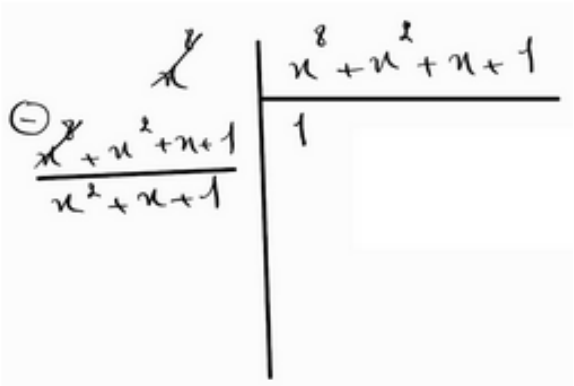

Figure 2: Division de $x^{\deg(\text{polynome générateur})} \cdot \text{data\_in}$ avec data_in $= X^0$

On obtient le pour le premier vecteur le CRC suivant : 0b0000_0111 $\Rightarrow$ **0x07**.

Le second vecteur en hexadécimal est le suivant : 0x0000000002.

Pour obtenir le CRC à la main, nous devons diviser $x^{\deg(\text{polynome générateur})} \cdot \text{data\_in}$ par le polynome générateur. Pour le second vecteur, nous avons le polynome suivant : X^1 que l'on multiplie par X^8 et on divise ensuite par le polynôme générateur.

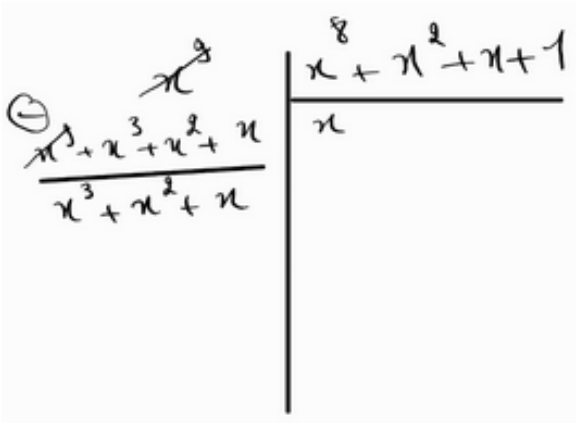

Figure 3: Division de $x^{\deg(\text{polynome générateur})} \cdot \text{data\_in}$ avec data_in $= X$

On obtient le pour le premier vecteur le CRC suivant : 0b0000_1110 $=>$ **0x0E**

Le CRC est le reste de la division euclidienne de X^9 par $x^8 + x^2 + x + 1$ qui est $X^3 + X^2 + X$ qui est équivalent à 0b0000_1110 en binaire et en hexadécimal nous obtenons 0x0E.

Nous utilisons le logiciel Questasim pour effectuer la partie simulation. Pour ce faire, nous lançons le fichier TCL : *simu.tcl*.

Dans le cadre de notre simulation, nous traitons un message de 40 bits. Il faut 40 coups d'horloge pour calculer le CRC. Chaque coup d'horloge a une durée de 20 ns.

2. Nous utilisons le logiciel Questasim pour effectuer la partie simulation.Pour ce faire, nous lançons un fichier TCL. Dans le cadre de notre simulation, nous traitons un message de 40 bits. Il faut 40 coups d'horloge pour calculer le CRC. Chaque coup d'horloge a une durée de 20 ns.

   Pour obtenir la durée totale nécessaire pour calculer le CRC, nous pouvons décomposer le processus comme suit :

   - Nous avons un message de 40 bits, ce qui signifie que nous devons effectuer 40 opérations.
   - En plus de ces opérations, nous devons tenir compte du temps nécessaire pour initialiser les bascules à l'aide de la commande "init", cela prend 20 ns.

   La durée totale requise pour obtenir le CRC, en partant du front montant de la commande "init", est de 20 ns (pour l'initialisation) + 40 * 20 ns (opération sur le message), soit un total de 820 ns.

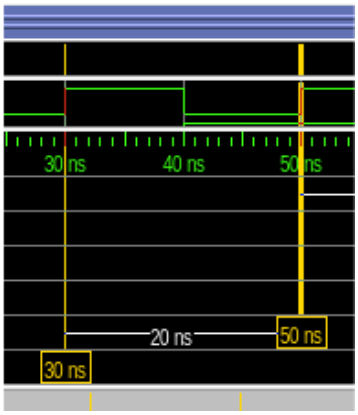

Figure 4: Période d'un coup d'horloge = 20 ns

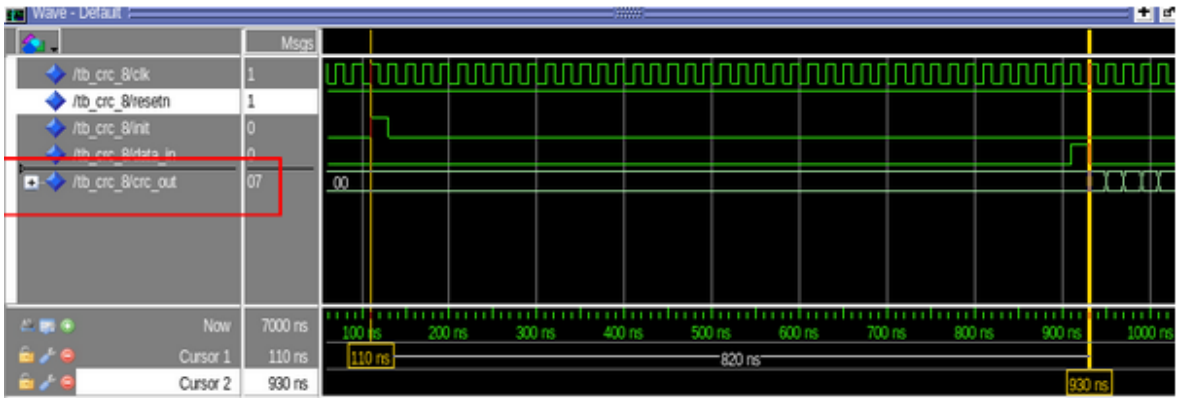

Figure 5: Vecteur numéro 1

Nous obtenons le CRC égale à 0x07 pour le vecteur numéro 1.

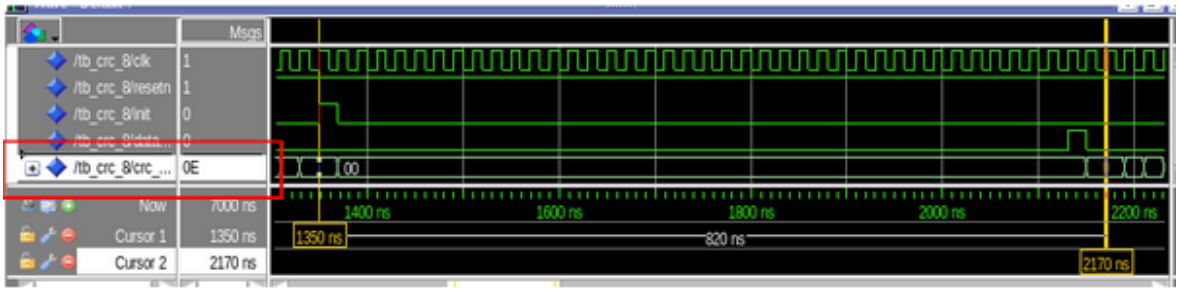

Figure 6: Vecteur numéro 2

Pour le vecteur numéro 2, nous obtenons le CRC suivant 0x0E.

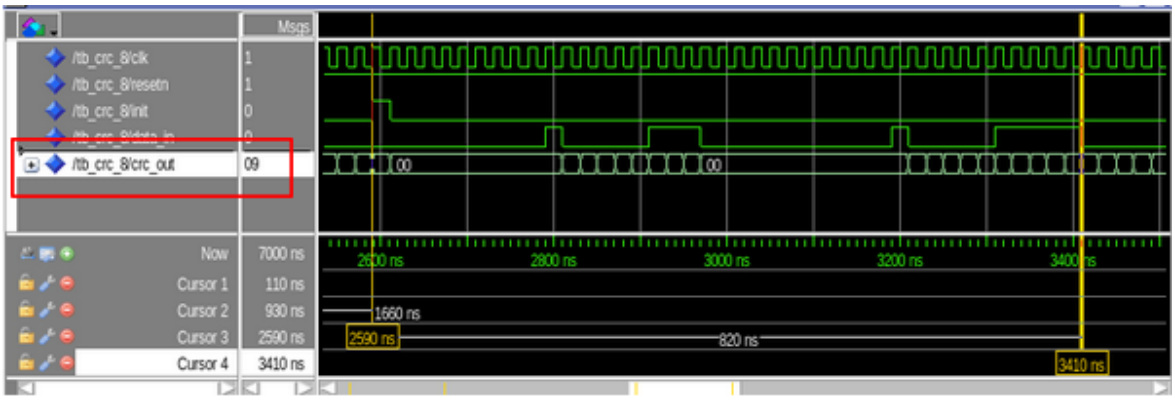

Figure 7: Vecteur numéro 3

Pour le vecteur numéro 3, nous obtenons le CRC suivant 0x09.

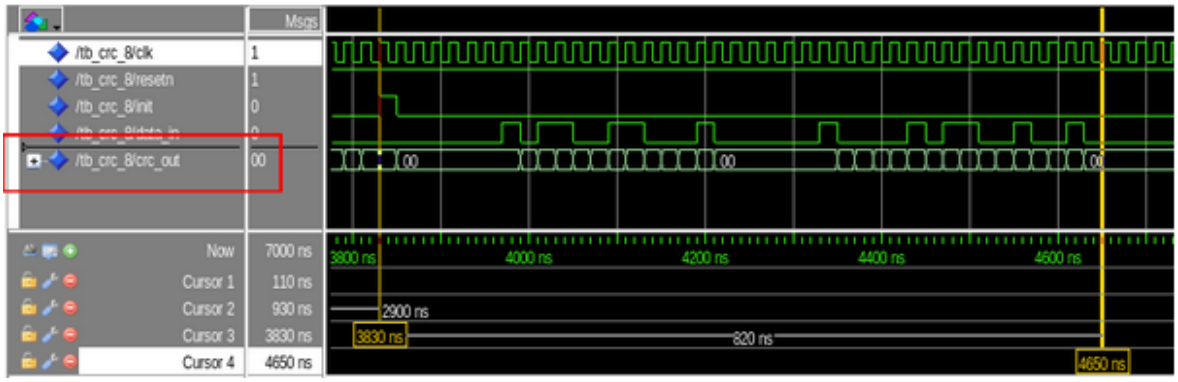

Figure 8: Vecteur numéro 4

Pour le vecteur numéro 4, nous obtenons le CRC suivant 0x00.

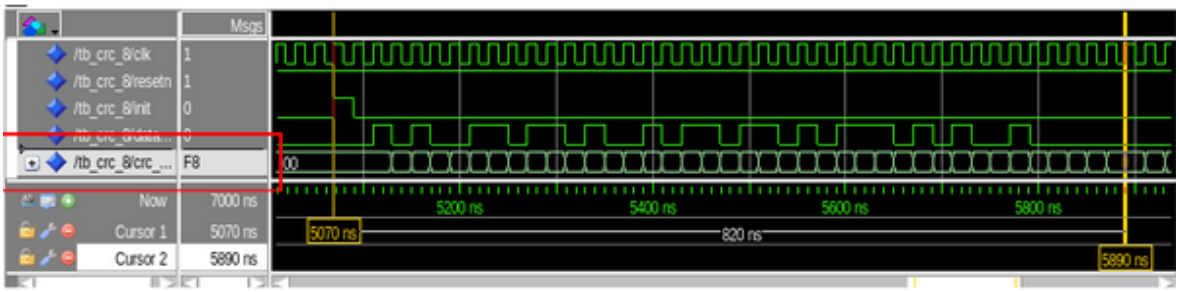

Figure 9: Vecteur numéro 5

Pour le vecteur numéro 5, nous obtenons le CRC suivant 0xF8.

3. Le CRC du cinquième vecteur est 0xF8.

Implémentation

1. En s'inspirant du fichier vu pour la démo de l'UART, nous créeons un fichier .xdc pour appliquer une contrainte de 100 MHz sur l'horloge.

   ## Clock signal
   set_property PACKAGE_PIN W5 [get_ports clk]
   set_property IOSTANDARD LVCMOS33 [get_ports clk]
   create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports clk]

2. Nous effectuons une implémentation pour une carte basys3. Nous créeons un projet CRC_8 dans le dossier "impl". Ce dossier contient tous les fichiers Vivado d'implémentation du projet. C'est dans ce dossier que nous allons chercher le fichier crc_8.vds (rapport de synthèse).

3. Nous analysons le rapport de synthèse pour vérifier que notre code à été correctement interprété. Pour cela, nous montrerons que notre implémentation ne posséde pas de latch, que nous avons le nombre correct de bascules, que ces dernières sont toutes initialisées de manière asynchrone et nous finirons par les warnings.

   Detailed RTL Component Info :
   +—XORs :
   2 Input 1 Bit XORs := 3
   +—Registers :
   8 Bit Registers := 1

+—Muxes :
2 Input 8 Bit Muxes := 1

Nous obtenons un registre de 8 bits. Trois XOR et un multiplexeur.

Le relevé RTL nous dresse le schéma suivant :

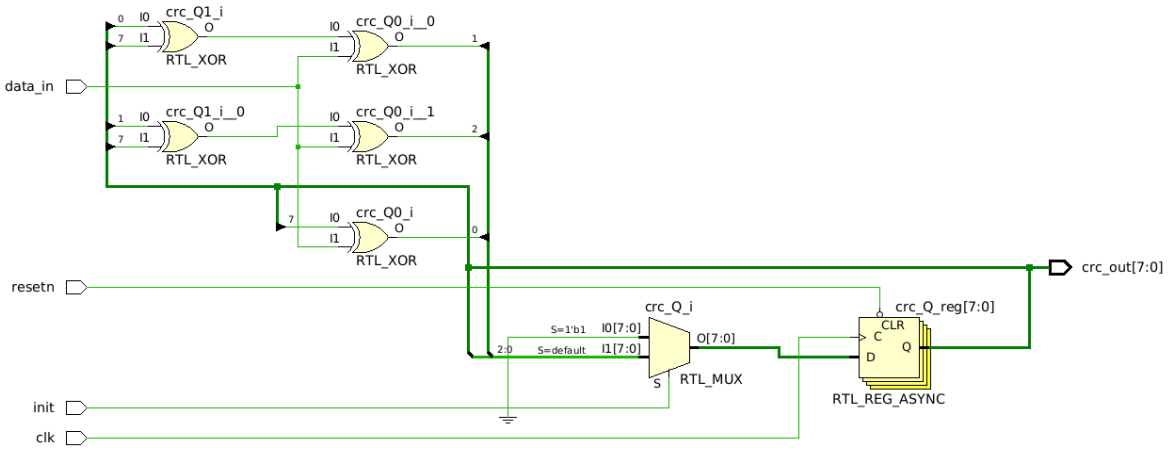

Figure 10: Schematic RTL Analysis

Nous retrouvons les huit bascules qui constitue le registre crc_Q. Le multiplexeur RTL_MUX permet de gérer la commande init afin d'initialiser les bascules. Nous retrouvons également, les XOR entre la DFF Q0 et la DFF Q1. Nous avons obtenu 5 XOR. Ce qui peut être du à l'optimisation de l'architecture RTL.

Après **Synthèse** nous avons :
Latch :

Il n'y a pas de latch (auncun LD*).
Il n'y a pas de bascule synchrone (FDRE).

```
Report Cell Usage:
+------+-----+------+
|      |Cell |Count |
+------+-----+------+
|1     |BUFG |     1|
|2     |LUT1 |     1|
|3     |LUT2 |     5|
|4     |LUT3 |     1|
|5     |LUT4 |     2|
|6     |FDCE |     8|
|7     |IBUF |     4|
|8     |OBUF |     8|
+------+-----+------+
```

Figure 11: Rapport d'usage

Bascule :

Nous avons huit bascules asynchrones d'après le rapport.

Report Cell Usage:

	Cell	Count
1	BUFG	1
2	LUT1	1
3	LUT2	5
4	LUT3	1
5	LUT4	2
6	FDCE	8
7	IBUF	4
8	OBUF	8

Figure 12: Rapport d'usage

Nous cherchons dans la documentation d'AMD Xilinx que veut dire FDCE.

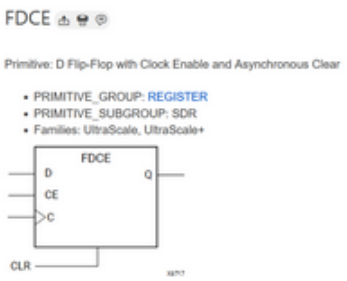

Figure 13: FDCE AMD Xilinx Documentation

Après lecture de la documentation, nous déduisons qu'il y a huit bascules. Elles sont toutes initialisées de manière asynchrone.

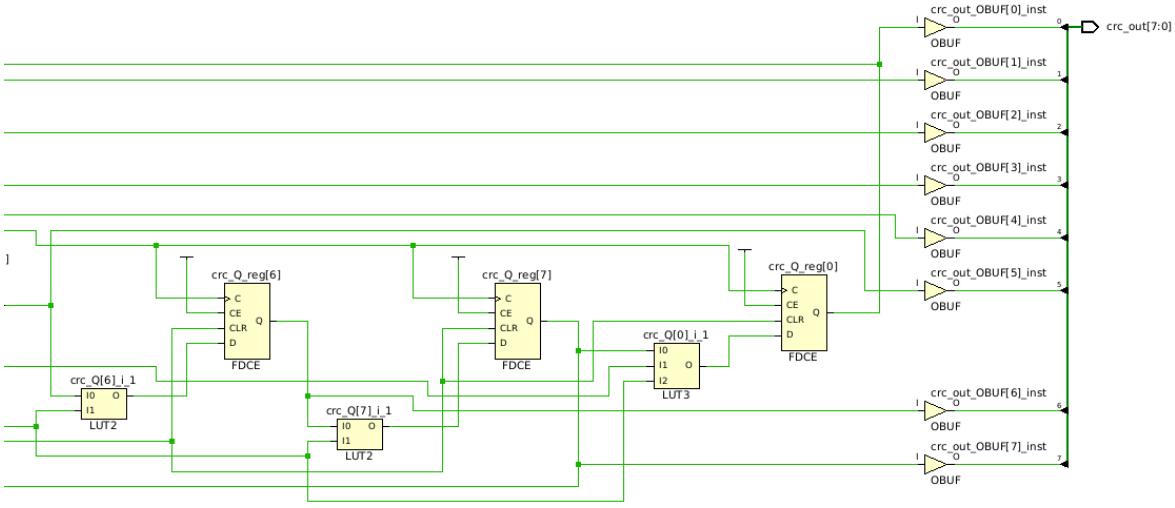

Figure 14: Schematic RTL Synthesis

Nous remarquons que nous avons huit bascules CRC_reg. Le signal prélevé à chaque sortie des bascules passe par un *output buffer* avant de sortir. Ce signal de sortie est crc_out.

Warnings :

Le rapport de synthèse nous informe qu'il y a 2 warnings.

WARNING: [Synth 8-7080] Parallel synthesis criteria is not met

Pour cet avertissement, notre implémentation est trop petite pour pouvoir être parallélisé.

WARNING: [Common 17-1361] You have specified a new message control rule that is equivalent to an existing rule with attributes ‹ -id {Synth 8-6859} -new_severity {ERROR} ‹ . The existing rule will be replaced

Cet avertissment est liée à Vivado.

4. Nous relevons les ressources après placement-routage. Pour ce faire, nous allons relevé les informations du rapport d'utilisation fourni par Vivado.

   Nous ouvrons le fichier Utilization - Place Design. Nous obtenons la distribution logique des slices :

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice	4	0	0	8150	0.05
SLICEL	2	0			
SLICEM	2	0			
LUT as Logic	5	0	0	20800	0.02
using O5 output only	0				
using O6 output only	1				
using O5 and O6	4				
LUT as Memory	0	0	0	9600	0.00
LUT as Distributed RAM	0	0			
LUT as Shift Register	0	0			
Slice Registers	16	0	0	41600	0.04
Register driven from within the Slice	4				
Register driven from outside the Slice	12				
LUT in front of the register is unused	11				
LUT in front of the register is used	1				
Unique Control Sets	1		0	8150	0.01

Figure 15: Distribution logique des slices

Le rapport de placement-routage nous informe qu'il y a 4 slices, 5 LUTs et 16 slices registers utilisés. Ce résumé concerne la version sans améliorations du code. Nous trouvons 16 slices registers qui correspond à 16 DFF. Nous nous questionnons du fait que nous obtenons le double après implémentation. (8 DFF pour la synthèse -> 16 DFF pour l'implémentation). Nous regardons le schèma RTL de l'implémantation. Il y a 16 DFF car il y a une réplique de chaque bascule avant le out buffer de la sortie. Nous ne savons pas pourquoi il y a une replication des bascules en sortie.

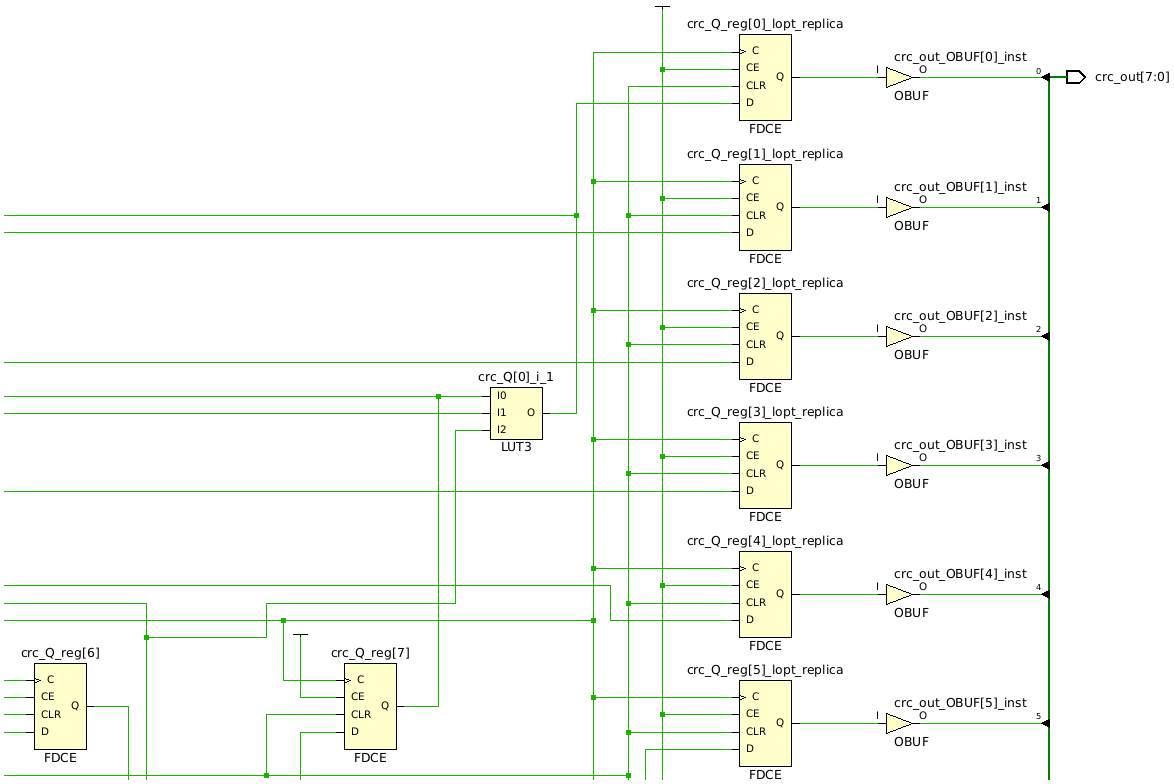

Figure 16: Schematic RTL Implementation

5. A partir du rapport de timings après placement-routage fourni par Vivado, nous extrayons l'information qui permet de déterminer la fréquence maximale de fonctionnement de notre composant. Pour cela, nous examinons le fichier suivant : crc_8_timing_summary_routed.rpt situé dans le répertoire impl/CRC/CRC.runs/impl_1/.

   Les lignes 149 à 156 indiquent que la contrainte d'horloge est de 10 ns soit 100 Mhz, on retrouve le nom de l'horloge sys_clk_pin donné lors du create_clock dans le fichier .xdc Nous relevons le résumé d'horloge.

```
| Clock Summary
| -------------

Clock        Waveform(ns)       Period(ns)      Frequency(MHz)
-----        ------------       ----------      --------------
sys_clk_pin  {0.000 5.000}      10.000          100.000
```

Figure 17: Résumé horloge

Le WNS est indiqué ligne 136 à 143 : 7,352 ns

```
| Design Timing Summary
| ---------------------

    WNS(ns)      TNS(ns)  TNS Failing Endpoints  TNS Total Endpoints      WHS(ns)      THS(ns)  THS Failing Endpoints  THS Total Endpoints
    -------      -------  ---------------------  -------------------      -------      -------  ---------------------  -------------------
      7.352        0.000                      0                   16        0.191        0.000                      0                   16
```

Figure 18: WNS

La fréquence maximale (à ne pas dépasser) pour cette implémentation est de $\frac{1}{(10-7.352)10^9} = 377$ MHz

Nous déterminons le chemin critique via le fichier texte : crc_8_timing_summary_routed.rpt

```
Max Delay Paths
--------------------------------------------------------------------------------
Slack (MET) :             7.352ns  (required time - arrival time)
  Source:                 crc_dff_reg[7]/C
                            (rising edge-triggered cell FDCE clocked by sys_clk_pin  {rise@0.000ns fall@5.000ns period=10.000ns})
  Destination:            crc_dff_reg[2]/D
                            (rising edge-triggered cell FDCE clocked by sys_clk_pin  {rise@0.000ns fall@5.000ns period=10.000ns})
  Path Group:             sys_clk_pin
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            10.000ns  (sys_clk_pin rise@10.000ns - sys_clk_pin rise@0.000ns)
  Data Path Delay:        2.314ns  (logic 0.668ns (28.868%)  route 1.646ns (71.132%))
  Logic Levels:           1  (LUT4=1)
  Clock Path Skew:        -0.025ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    4.839ns = ( 14.839 - 10.000 )
    Source Clock Delay      (SCD):    5.136ns
    Clock Pessimism Removal (CPR):    0.272ns
  Clock Uncertainty:      0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.000ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns

    Location             Delay type                Incr(ns)  Path(ns)    Netlist Resource(s)
-------------------------------------------------------------------    -------------------
                         (clock sys_clk_pin rise edge)
                                                    0.000     0.000 r
    W5                                              0.000     0.000 r  clk (IN)
                         net (fo=0)                 0.000     0.000    clk
    W5                   IBUF (Prop_ibuf_I_O)       1.458     1.458 r  clk_IBUF_inst/O
                         net (fo=1, routed)         1.967     3.425    clk_IBUF
    BUFGCTRL_X0Y0        BUFG (Prop_bufg_I_O)       0.096     3.521 r  clk_IBUF_BUFG_inst/O
                         net (fo=16, routed)        1.615     5.136    clk_IBUF_BUFG
    SLICE_X64Y81         FDCE                                    r  crc_dff_reg[7]/C
-------------------------------------------------------------------    -------------------
    SLICE_X64Y81         FDCE (Prop_fdce_C_Q)       0.518     5.654 r  crc_dff_reg[7]/Q
                         net (fo=3, routed)         1.021     6.674    crc_out_OBUF[7]
    SLICE_X64Y80         LUT4 (Prop_lut4_I1_O)      0.150     6.824 r  crc_dff[2]_i_1/O
                         net (fo=2, routed)         0.625     7.450    p_0_in[2]
    SLICE_X65Y80         FDCE                                    r  crc_dff_reg[2]/D
-------------------------------------------------------------------    -------------------
```

Figure 19: Chemin critique

Le chemin critique se trouve entre le bit 2 du registre crc_dff et le bit 7 du registre crc_dff. Ici il ne traverse qu'un seul LUT.

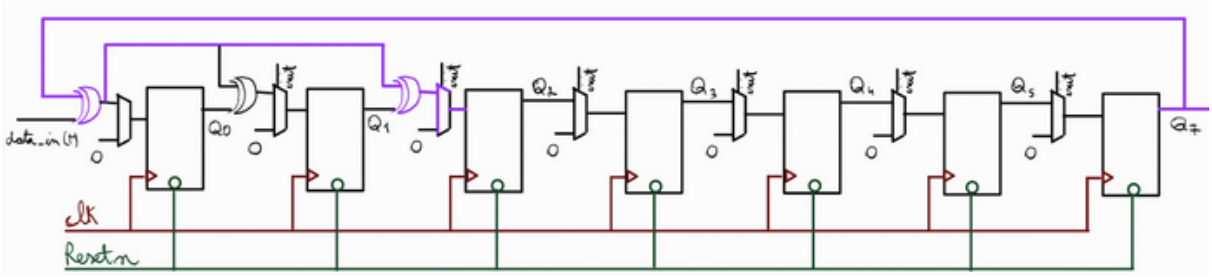

Figure 20: Schéma du chemin critique

Amélioration du code

L'objectif de cette partie est de rendre notre code facilement adaptable pour un autre polynôme de CRC (de 8 bits ou autre) et de n'avoir à modifier que la taille (nombre de bits) de crc_out et la valeur du polynôme.

1. On modifie le code pour décrire le polynome par une constante de type std_logic_vector. On utilise l'instruction for ... loop afin de décrire l'architecture. Tout d'abord, nous exprimons Q(0). Puis nous pouvons généraliser pour tout n différent de 0. C'est cette formule généraliser que nous mettons dans l'instruction for ... loop.
2. On utilise les attributs suivant. Ainsi, pour tout autre degré du CRC, il nous suffit de modifier uniquement la valeur du bit de poids fort, du port de sortie et la valeur (binaire) de la constante du polynôme.

3. Nous modifions le testbench pour réaliser des vérifications automatiques des résultats. Pour ce faire, nous créeons un tableau nommé *correct_result* composé des résultats. A l'aide d'une condition placée après la boucle d'envoi se trouvant dans la boucle i allant de 1 à 5. Nous testons la sortie. On n'oublie pas de remettre à zéro le crc_out avant de tester le resultat.
4. Nous réalisons une simulation post-implémentation avec timings depuis Vivado sur Questasim Advanced Simulator.

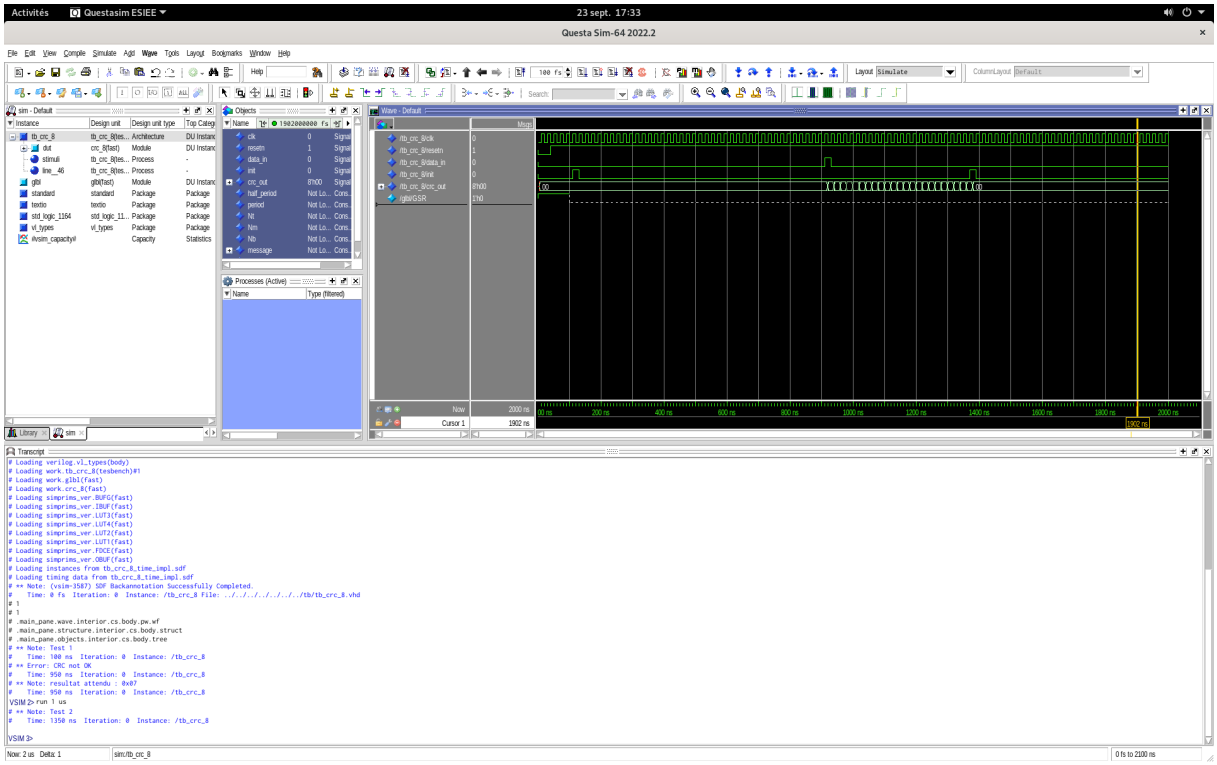

Figure 21: Simulation post-implémentation avec timings sur Questasim

En regardant la console Transcript de Questasim, nous relevons que nos tests sont erronés. Pour obtenir tous nos tests, nous utilisons la commande : *restart -f; run 7 us*

Relevé Questasim :

Note: (vsim-3587) SDF Backannotation Successfully Completed.
Time: 0 fs Iteration: 0 Instance: /tb_crc_8 File: ../../../../../../../tb/tb_crc_8.vhd
Note: **Test 1** # Time: 100 ns Iteration: 0 Instance: /tb_crc_8
Warning: **CRC not OK**
Time: 950 ns Iteration: 0 Instance: /tb_crc_8
Note: resultat attendu : 0x07
Time: 950 ns Iteration: 0 Instance: /tb_crc_8
Note: resultat obtenu : 0x0E
Time: 950 ns Iteration: 0 Instance: /tb_crc_8
Note: **Test 2**
Time: 1350 ns Iteration: 0 Instance: /tb_crc_8
Warning: **CRC not OK**
Time: 2210 ns Iteration: 0 Instance: /tb_crc_8
Note: resultat attendu : 0x0E
Time: 2210 ns Iteration: 0 Instance: /tb_crc_8
Note: resultat obtenu : 0x1C

Time: 2210 ns Iteration: 0 Instance: /tb_crc_8
Note: **Test 3**
Time: 2610 ns Iteration: 0 Instance: /tb_crc_8
Warning: **CRC not OK**
Time: 3470 ns Iteration: 0 Instance: /tb_crc_8
Note: resultat attendu : 0x09
Time: 3470 ns Iteration: 0 Instance: /tb_crc_8
Note: resultat obtenu : 0x12
Time: 3470 ns Iteration: 0 Instance: /tb_crc_8
Note: **Test 4**
Time: 3870 ns Iteration: 0 Instance: /tb_crc_8
Warning: **CRC OK**
Time: 4730 ns Iteration: 0 Instance: /tb_crc_8
Note: resultat obtenu : 0x00
Time: 4730 ns Iteration: 0 Instance: /tb_crc_8
Note: **Test 5**
Time: 5130 ns Iteration: 0 Instance: /tb_crc_8
Warning: **CRC not OK**
Time: 5990 ns Iteration: 0 Instance: /tb_crc_8
Note: resultat attendu : 0xF8
Time: 5990 ns Iteration: 0 Instance: /tb_crc_8
Note: resultat obtenu : 0xF7
Time: 5990 ns Iteration: 0 Instance: /tb_crc_8

Le test numéro 4 est un faux positif.

Nous regardons plus en détail la partie *Wave*.

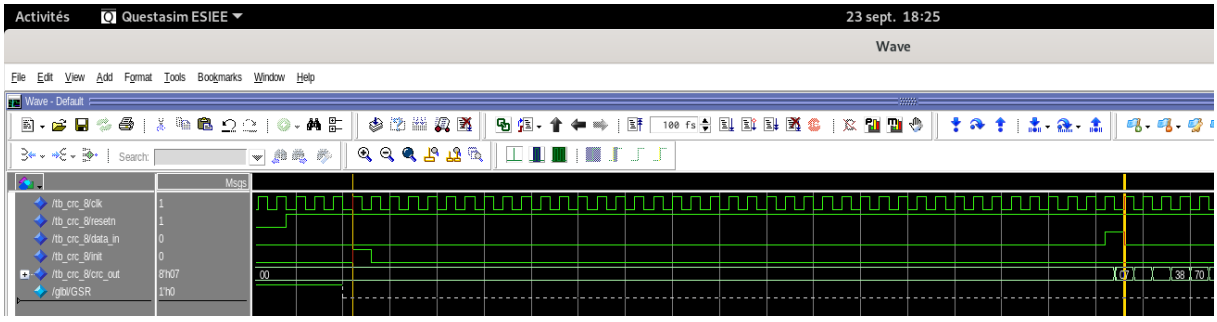

Figure 22: Simulation post-implémentation avec timings sur Questasim

Nous effectuant un agrandissement sur la zone du résultat du CRC.

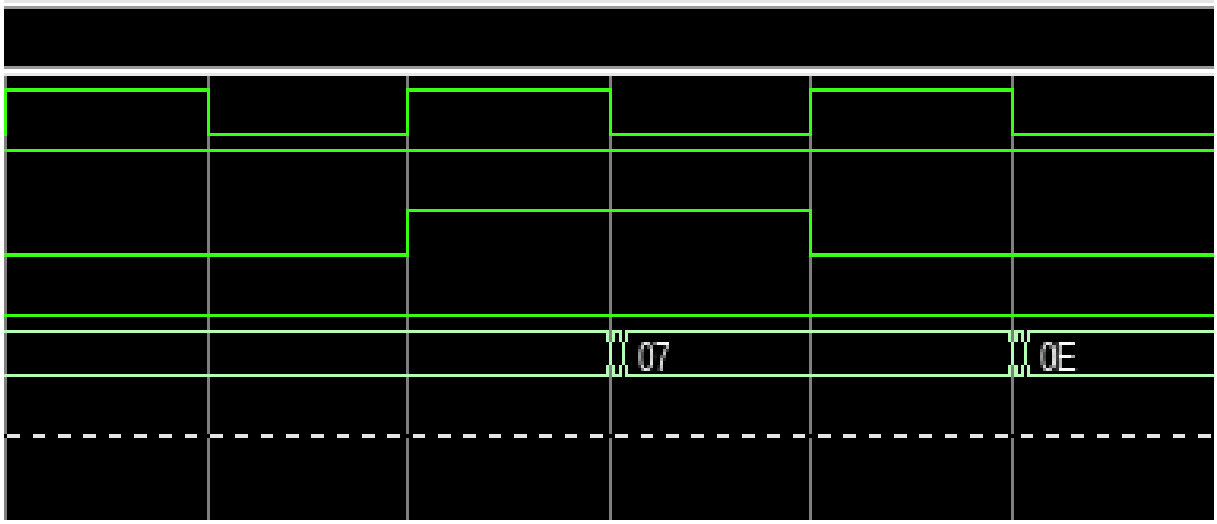

Figure 23: Agrandissment de la zone du résultat du CRC

Nous remarquons que la sortie du registre crc_out ne varie pas au moment du front montant. Le décalage est peut-être du aux portes, au routage et aux buffers de sortie. La simulation post-implémentation timings est une simulation se rapprochant le plus de la réalité.