Projet : Amélioration de l'asservissement de visée laser

Note d'application

Conception du correcteur RST en VHDL

Élève :
Mathis Pascal

Enseignant Référent :
Jacques Laffont
Client :
JTL Electronique

Table des matières

Table des figures

Liste des tableaux

Listings

Introduction

Cette note d'application a pour but de détailler le fonctionnement du correcteur RST codé en VHDL ainsi que son utilisation en mode Debug. Ainsi, l'ensemble des éléments constituant ce correcteur sera explicité.

Dans un premier temps, le bloc réalisant le calcul du correcteur seul sera présenté, ensuite l'instanciation avec la carte et le mode debug sera présenté.

I Correcteur RST

L'ensemble des codes utilisés pour la construction du correcteur sont présents dans le dossier Rebuilt_RST sur la forge.

Un correcteur RST est un correcteur utilisé en automatique pour réaliser l'asservissement de moteur, le schéma de principe est le suivant :

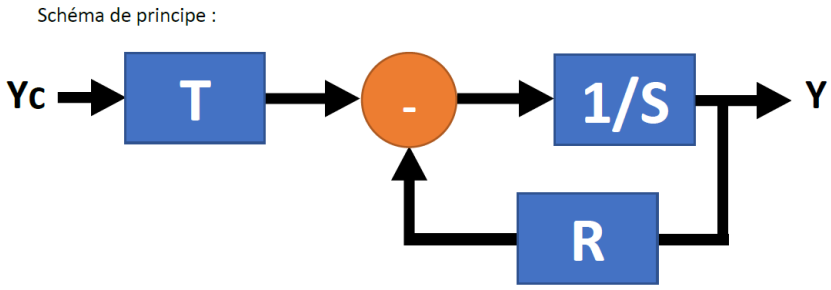

FIGURE I.1 – Schéma Correcteur RST

Pour la réalisation de ce correcteur, chaque étape est décomposée en petite partie afin de pouvoir tester au fur et à mesure de la construction du correcteur permettant de s'assurer de son fonctionnement.

La première étape est de réaliser les blocs R, S et T, réalisant les calculs.

Chaque bloc est composé de coefficients qui représentent, dans notre cas, le fonctionnement de notre galvanomètre.

Derrière chaque bloc se cache une équation mathématiques qui pour les blocs R et T sont identiques, cependant en VHDL il n'est pas possible d'inclure directement l'équation.

Ainsi, des registres sont utilisés pour stocker les valeurs puis les blocs multiplient leurs coefficients aux entrées du bloc ainsi qu'aux précédentes entrées stockées dans les registres. Ensuite, la somme de toutes ces multiplications est effectuée. Voici un schéma représentant ces calculs :

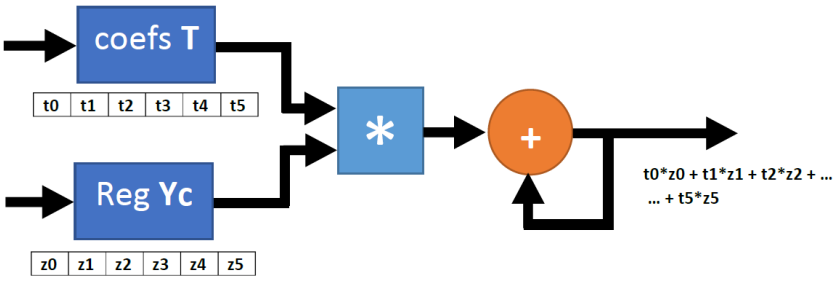

FIGURE I.2 – Schéma Bloc T

Ainsi, ce bloc réalise la somme de la multiplication des coefficients avec les entrées.

Pour le bloc S, c'est légèrement différent, car une division est compliquée en vhdl, donc plutôt que de faire 1/S, une soustraction est utilisé à la place de l'addition.

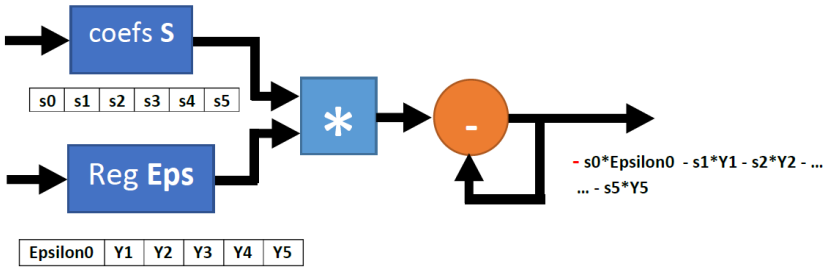

FIGURE I.3 – Schéma Bloc S

En fait le schéma est le même à la soustraction prêt. L'opération réalisée est bien 1/S, c'est à dire

$$S = coefficient0 * epsilon - \sum(coefficient * Y) \quad (1)$$

Avec epsilon qui représente la sortie du bloc T - la sortie du bloc R. Cependant, il faut ajouter un "-" devant le premier coefficient pour rendre le premier cycle de calcul valable sinon le premier résultat sera inversé.

Dans ce projet, l'ordre du système des années a été conservé, c'est-à-dire un ordre 6, ainsi chacun des blocs aura 6 coefficients.

De plus, à l'intérieur du correcteur les calculs sont faits avec des nombres flottant de 32 bits. Ainsi les calculs seront plus précis mais cela implique qu'il faudra donc convertir les variables pour pouvoir les interfacer avec la carte.

1 Composants Primaires

Dans un premier, les blocs de calcul (MAC) sont réalisés. Pour cela plusieurs composant élémentaires sont nécessaires notamment pour réaliser les additions et multiplication.

Afin de réaliser les opérations, des blocs IP sont utilisé, ces fonctions sont générées par altera.

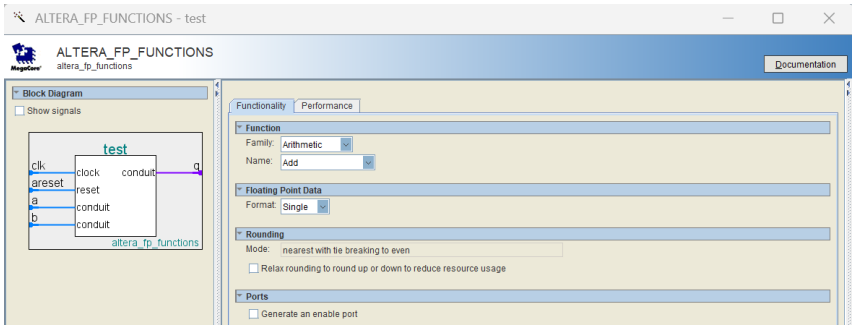

FIGURE I.4 – Bloc IP

Sur cette fenêtre, il est possible de choisir l'opération effectuée ainsi que la taille des opérandes, de plus dans l'onglet performance, la fréquence peut être rentrée afin de s'assurer de la cohérence avec la réalité. Dans notre cas, la fréquence est de 60MHZ, ainsi pour chacune des opérations, il faudra 3 cycles d'horloge pour produire un résultat.

Il ne reste qu'un composant pour pouvoir réaliser les blocs MAC de calcul, ce composant est très simple, il s'agit d'un accumulateur qui va simplement stocker le résultat pour pouvoir lancer l'addition suivante.

```
1  library ieee;
2  Use IEEE.STD_LOGIC_1164.all;
3  use IEEE.NUMERIC_STD.all;
4
5  entity mux_accu is port(
6    init : in std_logic;                -- '0' => s = e / '1' => s = x
       "00000000"
7    e : in std_logic_vector(31 downto 0);    -- entree (sortie de l'
       additionneur)
8    s : out std_logic_vector(31 downto 0));    -- sortie (retour    l'
       entree de l'additionneur)
9  end entity;
10
11 architecture ARCH_mux_accu of mux_accu is
12
13   begin
14
15     s <=
16     x"00000000" when init = '1' else  -- initialisation de l'
       accumulateur
17     e;
18
19 end architecture;
```

Listing 1 – Accumulateur

2 Réalisation MACs

Ces composants sont ensuite combinés pour réaliser les blocs de calcul, pour cela block diagram de Quartus est utilisé.

Pour les blocs R et T :

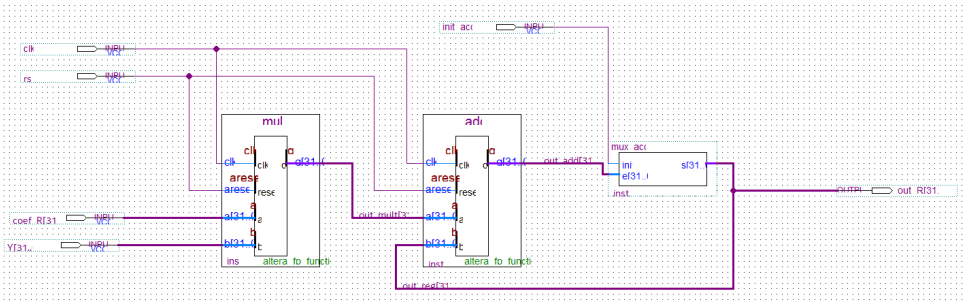

FIGURE I.5 – MAC R et T

Ainsi, ce bloc permet de faire la multiplication des entrées puis ajoute ce résultat à la somme des précédentes multiplications grâce aux mux_accu.

Un bloc similaire est réalisé pour le calcul de S, cette fois avec une soustraction.

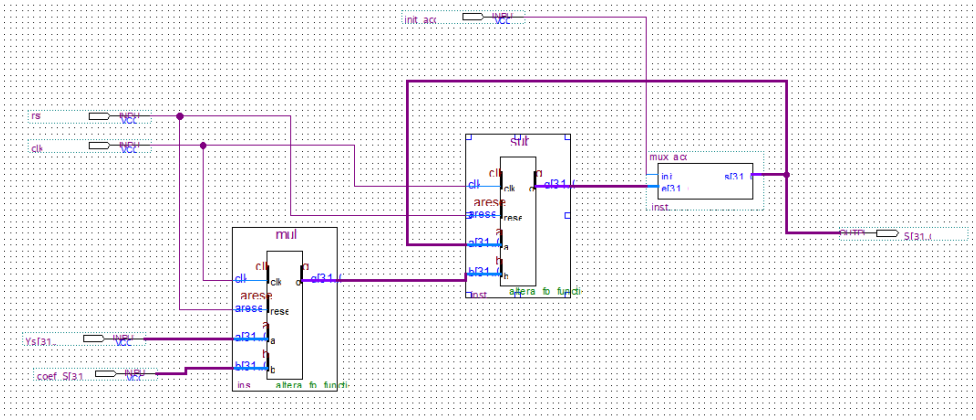

FIGURE I.6 – MAC S

Pour ce bloc, le mux est placé au dessus car dans l'équation, il faut faire la somme actuelle - la multiplication du nouveau coefficient. C'est pourquoi il faut un - devant le premier coefficient, car le mux est initialisé à 0 et de fait le premier résultat sera l'opposé de celui attendu si le coefficient n'est pas inversé.

L'avantage de découper la construction en petit, est que le fonctionnement de chaque bloc peut être validé. Ainsi après chaque étape un testbench est écris permettant de tester les blocs.

Un testbench est un code vhdl, dans lequel, les signaux d'entrée sont contrôlés, ce testbench est ensuite mis dans ModelSim qui va simuler notre design.

Par exemple pour le mac R voici un exemple de simulation.

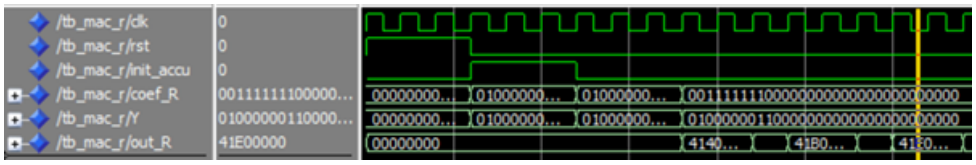

FIGURE I.7 – Simulation MAC R

Ces simulations permettent de vérifier si les calculs correspondent à la théorie. Les calculs sont faits en flottant il faut donc les convertir c'est pourquoi les résultats sont affichés en hexadécimal. Cependant, le but est ici est simplement d'expliquer la démarche, pas de vérifier les résultats. L'ensemble des testbench sont présent sur la forge, ainsi qu'un code scilab permettant de faire la conversion entre flottant et décimal.

Une fois le bloc validé par la simulation, il est possible depuis Quartus de générer le code associé au bloc diagram.

Cependant ce bloc ne réalise que les calculs, et pour l'instant les coefficients et les signaux d'entrée sont fournis par le testbench, ainsi la prochaine étape est d'ajouter des registres afin de stocker les différents coefficients.

3 Création Blocs séparés

Les MACs précédemment conçus sont repris auquel sont ajoutés les registres pour créer les trois blocs séparément. Les registres pour les coefficients et pour les entrées sont similaires, ils possèdent deux signaux en entrée, un signal "load" et un signal "shift".

Pour les coefficients, un identifiant est associé à chacun d'entre eux selon le tableau suivant :

Coefficient	Id_coef (decimale)
S0	131072
S1	131073
S2	131074
S3	131075
S4	131076
S5	131077
R0	65536
R1	65537
R2	65538
R3	65539
R4	65540
R5	65541
T0	196608
T1	196608
T2	196608
T3	196608
T4	196608
T5	196608

FIGURE I.8 – Tableau Id coefficients

Lorsque Load est à un si l'identifiant correspond le coefficient est chargé. Enfin le signal shift permet de décaler les coefficients pour faire les calculs avec chacun d'entre eux.

```
1  library ieee;
2  Use IEEE.STD_LOGIC_1164.all;
3  use IEEE.NUMERIC_STD.all;
4
5  entity reg_coef_T is -- ordre 6
6    port(
7    clk : in std_logic;
8    shift : in std_logic;                    --decalage dans le tableau
9    load : in std_logic;                       -- 1 => remplissage du tableau
       // 0 => ignore les entrees
10   val_coef : in std_logic_vector(31 downto 0);     --coef en entree
11   id_coef  : in std_logic_vector(31 downto 0);     --identification du
      coef (R, S ou T ?)
```

```
12    s : out std_logic_vector(31 downto 0));        --valeur de fin du
      tableau
13  end entity;
14
15  architecture ARCH_reg_coef_T of reg_coef_T is
16
17    type pattern_array is array (0 to 5) of std_logic_vector(31 downto 0);
        --Declaration d'un tableau de n elements
18     signal z : pattern_array;
19    --   | t0 | t1 | t2 | t3 | t4 | t5 |
20
21    begin
22      process (clk,shift,load,id_coef)
23        begin
24          if rising_edge(clk) then
25
26            if load = '1' then  -- Entree des coefs dans la memoire
27
28              ------------Coef R ------------
29              if id_coef = x"00030000" then --r0
30                z(0) <= val_coef;
31              end if;
32              if id_coef = x"00030001" then --r1
33                z(1) <= val_coef;
34              end if;
35              if id_coef = x"00030002" then --r2
36                z(2) <= val_coef;
37              end if;
38              if id_coef = x"00030003" then --r3
39                z(3) <= val_coef;
40              end if;
41              if id_coef = x"00030004" then --r4
42                z(4) <= val_coef;
43              end if;
44              if id_coef = x"00030005" then --r5
45                z(5) <= val_coef;
46              end if;
47
48            else
49
50              if shift = '1' then
51                for j in 1 to 5 loop
52                  z(j-1) <= z(j);
53                end loop;
54                z(5) <= z(0);    -- Bouclage du tableau
55              end if;
56
57            end if;
58          end if;
59      end process;
60
61      s <= x"00000000" when load = '1' else z(0);      -- ecriture de la
      sortie (r0, r1, ...)
62
63  end architecture;
```

Listing 2 – Registre à décalage Coefficient

En ce qui concerne le registre pour les entrées, lorsque le signal lorsque load est à 1, les anciennes valeurs sont décalées et la nouvelle entrée est chargé afin de garder l'ordre chronologique, pour le shift, c'est le même principe.

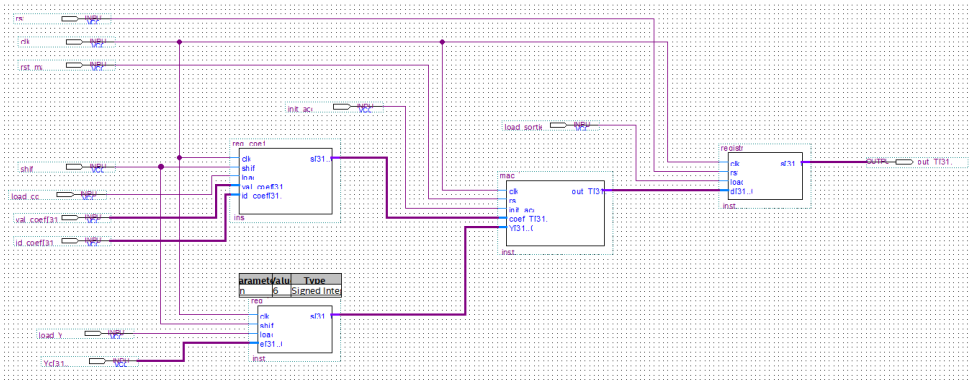

FIGURE I.9 – Blocs avec les registres

En sortie, un composant "registre" est codé, ce composant stocke le résultat lorsque que son load est à 1.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY registre IS
    PORT(
        clk        : IN std_logic;
        rst        : IN std_logic;
        load     : IN std_logic;
        d          : IN std_logic_vector(31 DOWNTO 0);
        s          : OUT std_logic_vector(31 DOWNTO 0)
    );
END registre;

ARCHITECTURE behavior OF registre IS
  signal q : std_logic_vector(31 downto 0);
BEGIN
    PROCESS(clk, rst)
    BEGIN
        IF rst = '1' THEN
            q <= (others => '0');

        ELSIF rising_edge(clk) THEN

            IF load = '1' THEN
                q <= d;
            END IF;
        END IF;
    END PROCESS;

  s <= d when load = '1' else q;

END behavior;
```

Listing 3 – Registre à décalage Coefficient

Les registres pour les coefficients et de sortie sont les mêmes pour les trois blocs (aux identifiants près).
Cependant, pour le bloc S, le registre pour les entrées est légèrement différent. Pour ce registre, lorsque load est à 1, il va charger deux entrées, la nouvelle valeur de Y et la valeur de epsilon (soustraction des deux autres blocs).

Un testbench est généré, ce testbench va contrôler les signaux de chargement et de décalage en tenant compte du temps nécessaire au composant IP pour faire le calcul afin que le bloc produise le bon résultat. Ainsi, chaque bloc peut être testé séparément avant l'assemblement du correcteur complet.

En réalité, chacun des blocs doit être autonome, ainsi chaque bloc va être associé à un composant appelé bloc de commande.

4 Bloc de commande

Ainsi le testbench précédant est repris pour coder un bloc de commande, ce bloc va générer les signaux de chargement et de décalage des registres aux bons moments pour que le bloc produise le bon résultat.

Afin de contrôler les timings, des compteurs sont utilisés. Un exemple de code pour le bloc de commande du bloc R est présent en annexe.

5 Blocs complets

Ce bloc de commande est ajouté pour chacun des blocs, cela permet de rendre les blocs autonome.
Voici, par exemple le bloc T complet :

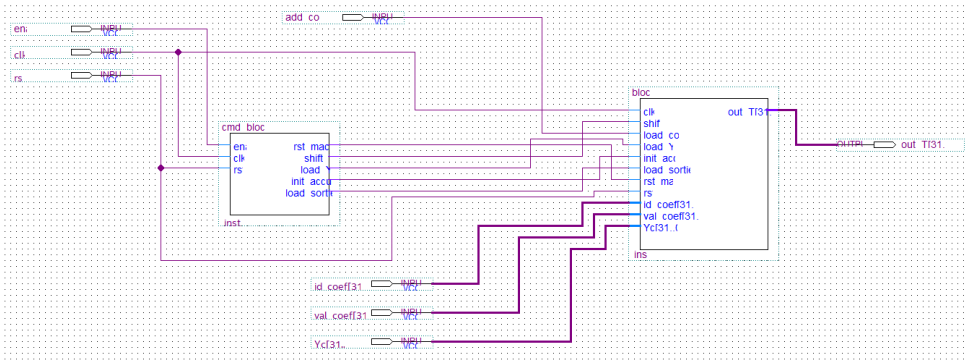

FIGURE I.10 – Bloc T complet

Ainsi, ce bloc comprend deux parties, le bloc de commande qui permet de contrôler le bloc de calcul. Le bloc est similaire au bloc T.

Cependant, le bloc S comporte une légère différence, en effet comme indiqué dans la première partie, les calculs sont fait en flottant 32 bits, cependant la carte la carte ne supporte uniquement des entiers 17 bits.

Ainsi, un convertisseur de flottant vers entier (float2fix) est placé en sortie du bloc S, car cette sortie est la commande qui sera appliqué aux galvanomètres. Ce convertisseur est réaliser grâce au fonctions IP de chez altera comme pour les additionneurs.

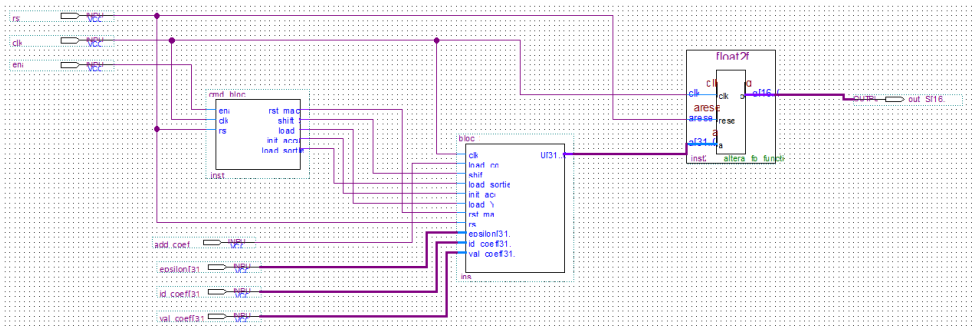

FIGURE I.11 – Bloc S complet

Désormais, chaque bloc est autonome, afin d'élargir les tests, ces derniers sont fait sur ModelSIm avec des testbench comme précédemment, mais également en réel. En effet, les blocs sont implémentés seuls sur la carte, puis avec signalTap (outil de Quartus), les résultats des blocs sont vérifiés directement sur la carte.

Cela à permis de valider le fonctionnement de ces blocs séparément.

La dernière étape consiste a rassembler ces trois blocs pour former le correcteur RST.

6 Correcteur RST complet

Les trois blocs sont placés conformément au schéma figureI.1. Ensuite, le convertisseur inverse, d'entier vers flottant est placé en entrée du bloc R pour recevoir les données provenant des galvanomètres. Enfin, un bloc de contrôle est créé, ce bloc va activer les différent bloc au bon moment en tenant compte du temps nécessaire de chaque bloc pour produire un résultat et la réalisation de la soustraction.

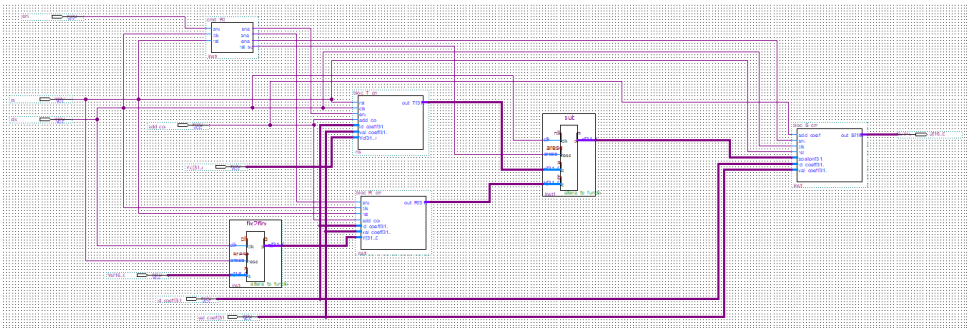

FIGURE I.12 – Correcteur RST

Le code de ce diagramme peut être générer. Ensuite, il faut implémenter ce correcteur sur la carte pour pourvoir valider son fonctionnement. Le code généré par quartus est fournit en annexe.

II Implémentation RST

A ce niveau le correcteur est codé cependant, il faut encore l'implémenter sur la carte.
Étant donné que l'interface des années précédentes est conservées, il suffit de remplacer les anciens codes par les nouveaux.

Ensuite il faut bien vérifier si l'instanciation avec le fichier new_component.vhd.
Ce fichier réalise l'interface entre le FPGA et le NIOS, c'est dans ce code que les registres sur lesquels les entrées sont envoyées sont définis.

Enfin dernier point, il faut ajuster le fichier debug_enable_RST. Car ce fichier lance les nouveaux cycles de calcul, il faut donc ajuster le compteur pour que celui-ci corresponde avec notre RST, dans notre cas un calcul prend environ 90 cycles, soit 1,5 us à notre fréquence.

Enfin, le projet peut être compilé :

Total logic elements	10,596 / 24,960 (42 %)

FIGURE II.1 – Place occupée RST

Avant cette architecture, un correcteur utilise 42% de la place disponible sur le FPGA, ainsi inclure un deuxième correcteur pour aiguiller la voie Y est possible.

Un mode debug a été codé, ce mode permet de réaliser les calculs cycle par cycle, en effet un cycle de calcul en mode normal est trop rapide.

Le mode debug nécessite d'inclure de nombreux registres pour pouvoir choisir le mode par exemple. Cela ajoute du temps de développement mais cela permet également de pouvoir vérifier et débugger le correcteur cycle par cycle.

III Utilisation du mode Debug

Pour utiliser le mode debug il faut d'abord activer les bons paramètres dans les registres.

Les registres sont définis dans le fichier new_component, puis ils sont explicités dans la note d'application de Mr Régent.

Par exemple, pour envoyer les coefficient, il faut écrire l'identifiant (en décimal) sur le registre 9 et la valeur du coefficient (en décimal) sur le registre 10. Une fonction scilab permet de faire la conversion de flottant vers décimal.

Sur la forge, dans le fichier Testbench_verif_RST des fonctions add_coeff sont déjà codées. De plus il y a aussi un fonction cycle_RST_debug qui permet de lancer plusieurs cycles de calcul.

Cependant voici la procédure pour lancer le mode debug cycle par cycle :

```
-> Faire nc_port = <num\_port>
-> Lancement du script Testbench_verif_RST
-> Lancement la fonction init_debug, cette fonction met en place le mode debug
-> Ecriture valeur consigne : nc_write(12, <valeur_consigne_en_décimal)
-> Ecriture valeur entrée bloc R : nc_write(13, <valeur_consigne_en_flottant)
-> cycle_RST() pour lancer un cycle de calcul
```

Enfin, pour visualiser le résultat, il y a deux possibilités :
->lire la valeur de la sortie du RST depuis les registre (registre 13) -> Utiliser signal_tap

Signal_tap est un outil de Quartus, l'avantage de cet outil est qu'il permet de visualiser les signaux internes du correcteur et donc de debugger plus facilement.

IV Conclusion

En conclusion, cette note d'application présente la façon dont le correcteur RST est codé. De plus cette note d'application permet de comprendre les étapes de conception de ce dernier, enfin le mode debug est présenté ainsi qu'un exemple d'utilisation.

Les résultats des tests effectués avec ce correcteur sont présenté dans le rapport.