Électronique numérique 2 (MNE M1S2)

Rapport : *Implémentation de systèmes numériques sur FPGA à l'aide du langage VHDL - Générateur des termes Fibonacci*

Écrit par : Justin Silver

Groupe B (TP)

Fichier électronique : *rapport_num_M1S2_SILVER.pdf*

Résumé

Pour le master Micro et nano-électronique semestre 2 de l'année 2020-2021, voici la réalisation d'un générateur des termes Fibonacci sur FPGA (Cyclone V) se situait sur la carte de développement GX-Starter Kit. Les termes sont générés à la cadence d'environ 1 Hz et puis sont affichés en BCD sur des afficheurs 7 segment LED (4 digits). Pour détecter la génération des termes erronées résultant des overflow, une communication UART est employée entre la carte de développement et un ordinateur hôte (qui tourne un terminal série). Les termes non-erronées engendreront un envoi de la caractère « B » et les termes erronées engendreront un envoi de la caractère « e ». L'utilisateur peut sélectionner le nombre de termes Fibonacci à générer, redémarrer le système via un reset, et basculer entre 9600 baud ou 19200 baud pour le module UART.

Table des matières

Résumé 2
Table des matières 3
Figures 3
Tableaux 3
Annexes 4
GitHub – suivi du développement 4
Introduction, vue du système 4
Modules 7
Module digit_types 7
Module baud_gen 7
Module bin2bcd 8
Module blinky 8
Module char_select 8
Module edge_divider 9
Module fibonacci_gen 9
Module ledseg_decoder 9
Module n_ledseg_decoder 9
Module system_control 9
Module uart_control 10
Module uart_transmit 10
Testbenches 10
Difficultés rencontrées 12
Conclusion 12
Références 13

Figures

Figure 1. Schéma bloc du système 5
Figure 2. Calcul pour la génération du baud-rate 7
Figure 3. Exemple de calcul pour l'algorithme "Double Dabble" 8
Figure 4. Sélection de caractère (ASCII) 8
Figure 5. Générateur des décodeurs ledseg 9
Figure 6. Testbench simple pour le module "edge_divider" 11
Figure 7. Testbench plus complexe pour le module "bin2bcd" 11

Tableaux

Tableau 1. Description des blocs du système 7

Annexes

Annexe 1. Rapport du générateur de nombres Fibonacci (2020) – Justin SILVER 14
Annexe 2. Module digit_types 18
Annexe 3. Module baud_gen 19
Annexe 4. Testbench pour baud_gen 21
Annexe 5. Module bin2bcd 22
Annexe 6. Testbench pour bin2bcd 24
Annexe 7. Module blinky 26
Annexe 8. Module char_select 27
Annexe 9. Module fibonacci_gen 28
Annexe 10. Testbench fibonacci_gen 30
Annexe 11. Module edge_divider 32
Annexe 12. Testbench edge_divider 33
Annexe 13. Module ledseg_decoder 34
Annexe 14. n_ledseg_decoder 35
Annexe 15. Module system_control 36
Annexe 16. Testbench system_control 38
Annexe 17. Module uart_control 39
Annexe 18. Testbench uart_control 41
Annexe 19. Module uart_transmit 42
Annexe 20. Testbench uart_transmit 44

GitHub – suivi du développement

Pour les lecteurs intéressés, le développement du projet peut être investigué sur ma page GitHub : www.github.com/sir-drako/fpga-fibonacci.

Introduction, vue du système

Dès le départ du projet, une importance a été mise sur la modularité et sur la testabilité du système. Le schéma bloc du système global (réalisé en Quartus) est inclus ci-dessous :

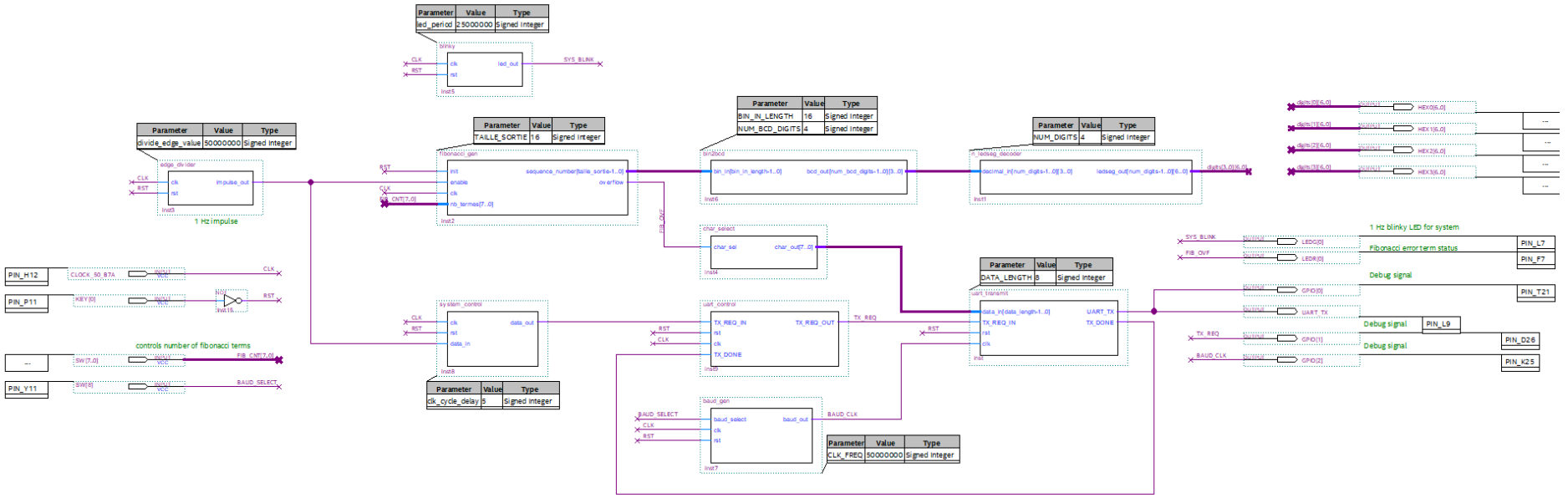

Figure 1. Schéma bloc du système

Les différents blocs du système et leurs rôles sont résumés dans la table suivante :

Bloc	Description
edge_divider	Ce bloc sert comme source d'impulsion à la cadence de 50 MHz. Le pulse dure un cycle d'horloge par défaut mais son rapport cyclique est paramétrable via un générique
blinky	Pour signaler le bon démarrage du système et pour donner une visualisation de la cadence du système, une LED verte clignote à la fréquence de 1 Hz. Sa période est paramétrable via un générique
fibonacci_gen	Générateur des termes Fibonacci. Son initialisation est effectuée via un signal de reset synchrone en entrée. Il nécessite également un enable synchrone avec un front montant pour générer le prochain terme. Le nombre de termes Fibonacci est déterminé à partir d'une byte en entrée. La taille du bus de sortie est paramétrable. Par défaut il est mis à 16 bits, en tenant compte du cahier de charges
system_control	Ce bloc gère le commencement de la communication UART en imposant un délai de l'impulsion venant du bloc « edge_divider ». Ce délai (en cycles d'horloges) est paramétrable. Il se situe entre les modules UART et le module d'impulsion pour éviter l'envoi d'un caractère au mauvais moment. Vu qu'il existe un délai de propagation du signal « overflow » venant du module « fibonacci_gen », il se peut que le signal d'overflow change d'état après le démarrage d'une transmission UART
bin2bcd	Un convertisseur des nombres binaires en BCD qui utilise l'algorithme « Double Dabble ». La taille du nombre binaire et le nombre de digits BCD en sortie sont paramétrables via des génériques
char_select	Un simple bloc « multiplexeur » qui sélectionne soit la valeur de la caractère ASCII « B » soit la valeur de la caractère ASCII « e » selon l'état de l'overflow du bloc générateur Fibonacci
uart_control	Ce bloc gère la détection de l'impulsion dans le 50 MHz « clock domain ». Il implémente aussi un protocole de « handshaking » avec le bloc UART responsable pour la transmission de données (ce bloc existe dans le baud rate « clock domain »). Finalement ce bloc de contrôle attend la remise à 0 de l'impulsion avant de recommencer une nouvelle transmission
baud_gen	Générateur du baud rate (9600 baud ou 19200 baud)
n_ledseg_decoder	Un décodeur à n digits BCD en donnée 7 segment LED (anode en commun). Par défaut ce bloc sort 4 digits de données LED segment pour les 4 digits présents sur la carte GX

uart_transmit	Cette partie du système « UART » sert à détecter une requête de transmission venant du domaine « 50 MHz », puis l'acquitter. Ce bloc gère ensuite la communication avec le récepteur UART (ordinateur hôte). Notamment, la transmission des bits START/STOP et des données ASCII sortant d'un registre à décalage

Tableau 1. Description des blocs du système

Modules

Les modules VHDL et leurs fonctionnalités variées sont présentés en plus de détails dans les sections prochaines. L'intégralité de leur code source se retrouve en annexe.

Module digit_types

Ce module (package) sert à facilement paramétrer les autres modules qui travaillent avec les données de type « digits » (digit de type BCD, digit de type segment LED). Dedans il existe des « unconstrained arrays » en jargon VHDL, ce qui signifie que ces arrays de type digit n'ont pas de longueurs définies. Elles sont plutôt définies lors d'une déclaration d'un objet ayant ce type.

Module baud_gen

L'horloge sortant de ce module est généré en comptant les cycles d'horloge d'entrée et en comparant le compte actuel avec le nombre de cycles nécessaire pour générer une fréquence d'horloge à 9600 baud ou à 19200 baud. La détermination de ce nombre se fait à partir de la fréquence d'horloge en entrée. Voici un explicatif pour comment calculer le nombre de cycles d'horloge nécessaire pour générer le baud rate ciblé.

```
-- to determine the count for obtaining the specified baud rate, the following
-- formula is used: CLK_FREQ/X = BAUD_RATE/2 => CLK_FREQ/(2*BAUD_RATE)

-- example: to have a baud rate of 9600 (9600 bits/s) with a CLK_FREQ of 50 MHz,
-- we have to count to 50*(10^6)/(19200), thus X ~= 2604. This means that every 2604 ticks,
-- the baud_out signal's state will toggle. In one second, there will be 50 million ticks, which means that
-- the baud_out signal would have toggled ((50*10^6)/2600) times, ~= 19200.
-- THUS, the frequency of the baud_out signal would be 19200/2, = 9600.

constant COUNT_BAUD_9600 : integer := CLK_FREQ/(2 * 9600);
constant COUNT_BAUD_19200 : integer := CLK_FREQ/(2 * 19200);
```

Figure 2. Calcul pour la génération du baud-rate

Module bin2bcd

Ce module gère la conversion des nombres binaires sortant du générateur des termes Fibonacci en N digits de nombres BCD. L'algorithme sélectionné est « Double Dabble », qui est une solution (dans cette application) purement combinatoire. Son implémentation nécessite des simples décalages, des additions, etc. Un exemple de calcul est détaillé ci-dessous [1].

The double-dabble algorithm, performed on the value 243_{10}, looks like this:

```
0000 0000 0000   11110011   Initialization
0000 0000 0001   11100110   Shift
0000 0000 0011   11001100   Shift
0000 0000 0111   10011000   Shift
0000 0000 1010   10011000   Add 3 to ONES, since it was 7
0000 0001 0101   00110000   Shift
0000 0001 1000   00110000   Add 3 to ONES, since it was 5
0000 0011 0000   01100000   Shift
0000 0110 0000   11000000   Shift
0000 1001 0000   11000000   Add 3 to TENS, since it was 6
0001 0010 0001   10000000   Shift
0010 0100 0011   00000000   Shift
   2    4    3
       BCD
```

Figure 3. Exemple de calcul pour l'algorithme "Double Dabble"

Module blinky

Ce module sert à signaler à l'utilisateur que le système s'est mis en route. Son fonctionnement est très similaire au module « baud_gen », car les deux cherchent à atteindre les mêmes objectifs : générer un signal numérique périodique à une fréquence donnée.

Module char_select

Ce module est construit comme un simple multiplexeur avec la syntaxe VHDL « with… else ». Il exploite le fait que le signal overflow venant du générateur pourrait être utilisé comme « signal de sélection » du multiplexeur. Évidemment, l'envoi d'un plus grand nombre de caractères nécessitera un système de sélection plus complexe.

```vhdl
architecture logic of char_select is
begin
    with char_sel select
        char_out <= x"42" when '0', -- the ASCII character 'B' represents 'BON'
                    x"45" when '1', -- the ASCII character 'E' represents 'ERREUR'
                    x"00" when others; -- we could add other chars if needed
end architecture;
```

Figure 4. Sélection de caractère (ASCII)

Module edge_divider

Ce module fonctionne très similairement aux modules « blinky » et « baud_gen ». Il faut simplement compter un certain nombre de cycles à l'aide d'une variable et une fois le compte est atteint, l'état de la sortie du bloc change. Dans ce cas, contrairement aux autres modules, la sortie passe à 1 pendant un seul coup d'horloge puis elle repasse à 0. Par conséquent, le module sort une impulsion de très courte durée, au lieu d'un signal périodique avec un rapport cyclique de 50%.

Module fibonacci_gen

Pour plus de détails sur le fonctionnement de ce module, le lecteur peut se référer au rapport dédié à sa réalisation [2]. Ce document est dirigé par le même auteur que ce rapport (Justin SILVER) et il est inclus en annexe.

Module ledseg_decoder

Avec un simple « case » syntaxe, le décodage des segments LED à partir des nombres BCD est atteint.

Module n_ledseg_decoder

Ce module est une extension du module ledseg_decoder qui permet de générer un nombre N de digits 7 segments (autrement dit, un nombre de digits paramétrable). Ceci est atteint en employant la syntaxe « generate » avec le composant « ledseg_decoder ».

```
build_n_digits : for i in 0 to (NUM_DIGITS - 1) generate
    -- dont need to do specify ledseg_decoder architecture
    -- (there's only one)
begin
    decoder : ledseg_decoder
    port map(
        decimal_in => decimal_in(i),
        ledseg_out => ledseg_out(i)
    );
end generate build_n_digits;
```

Figure 5. Générateur des décodeurs ledseg

Module system_control

Ce module fonctionne en machine d'état pour générer un « délai » d'impulsion pour le signal arrivant du module « edge_divider ». Ce délai est paramétrable.

Module uart_control

Ce module fonctionne en machine d'état pour communiquer avec le module « uart_transmit ».

En IDLE, il attend la réception de l'impulsion de l'edge divider.

En WAIT_ACK, il communique avec uart_transmit et attend son acquittement avant d'attendre la fin de la transmission. Il faut attendre l'acquittement du module uart_transmit avant d'immédiatement passer au prochain état car le même signal pour l'acquittement de transmission et de la fin de transmission est utilisé.

En WAIT_EOT, le bloc de contrôle a vérifié que le uart_transmit a démarré une nouvelle transmission et donc le bloc attend la fin de cette transmission.

Finalement, en WAIT_RESET ce bloc vérifie que le signal d'impulsion n'est pas actif. Ceci ne devrait jamais être le cas car le signal d'impulsion passe immédiatement à 0 un cycle d'horloge (fonctionnant à 50 MHz) plus tard. Par conséquent, cet état n'est pas strictement nécessaire dans cette application particulière. Cela dit, son inclusion permet de « découpler » le module uart du module edge_divider. En effet, un état explicit dédié à la remise à 0 du signal de demande de transmission dans le domaine de 50 MHz améliore la portabilité de ce module.

Module uart_transmit

Ce module fonctionne en machine d'état. Il complémente le module UART de contrôle. Ce bloc contient aussi un registre à décalage pour envoyer les bits les uns après les autres au récepteur UART se situant dans l'ordinateur hôte. Des bits de START et STOP sont intégrés dans les états IDLE, DATA_TX et END_TX mais cette implémentation pose quelques désavantages. Notamment, le nombre de START/STOP bits n'est pas paramétrable. Par conséquent, si plusieurs STOP bits sont nécessaires par le hardware du récepteur, il faut modifier le fonctionnement de ce module.

Testbenches

Pour vérifier le bon fonctionnement de tous les modules, le simulateur ModelSim qui fait partie du logiciel Quartus a été employé avec des testbenches VHDL. Pour la majorité des modules, ces testbenches ont été assez simplistes, car il fallait simplement observer l'évolution des signaux sortant des modules.

Par exemple, en tenant compte du fait que le module edge_divider ne contenait pas de signaux d'entrée (à part de l'horloge) son testbench consistait seulement des interconnexions du « Device Under Test (DUT) » et de ses vecteurs de tests pour son horloge (un signal périodique à 50 MHz) et pour son reset.

```
begin
    clk_process : process
    begin
        clk <= '0';
        wait for CLK_PERIOD/2;
        clk <= '1';
        wait for CLK_PERIOD/2;
    end process;


  inst_edge_divider : edge_divider
    generic map (
      divide_edge_value => divide_edge_value
    )

    port map (
        clk => clk,
        rst => rst,
        impulse_out => impulse_out
    );

    rst  <= '1',
            '0' after 15 ns;
```

Figure 6. Testbench simple pour le module "edge_divider"

En revanche, le testbench pour le convertisseur décimal-BCD a été plus compliqué, car celui-ci recevait des signaux en entrée. Le syntaxe « assert » a été utilisé pour à rapidement vérifier la conversion valide des nombres binaires en entrée, en évitant la nécessité de directement investiguer les signaux de sortie.

```
-- should return 4095
bin_in <= x"0fff";
wait for CLK_PERIOD*10;
assert bcd_out(3) & bcd_out(2) & bcd_out(1) & bcd_out(0) = x"4095" severity error;

-- should return 0
bin_in <= x"0000";
wait for CLK_PERIOD*10;
assert bcd_out(3) & bcd_out(2) & bcd_out(1) & bcd_out(0) = x"0000" severity error;

-- should return 2748
bin_in <= x"0abc";
wait for CLK_PERIOD*10;
assert bcd_out(3) & bcd_out(2) & bcd_out(1) & bcd_out(0) = x"2748" severity error;

-- should return 9999
bin_in <= x"270F";
wait for CLK_PERIOD*10;
assert bcd_out(3) & bcd_out(2) & bcd_out(1) & bcd_out(0) = x"9999" severity error;

-- should return 312
bin_in <= x"0138";
wait for CLK_PERIOD*10;
assert bcd_out(3) & bcd_out(2) & bcd_out(1) & bcd_out(0) = x"0312" severity error;
```

Figure 7. Testbench plus complexe pour le module "bin2bcd"

Le code source pour tous les testbenches se retrouve en annexe.

Difficultés rencontrées

- *Communication UART – non-envoi des caractères*

Ma première approche pour envoyer un caractère ne marchait pas, car je ne prenais pas en compte la différence d'horloges du module UART et du module « edge_divider ». En effet, il fallait réaliser un système d'acquittement pour garantir que le module UART pouvait démarrer une communication et signaler au bloc de contrôle qu'il est disponible pour un nouvel envoi.

- *État de système bloqué*

Le module de transmission UART pouvait envoyer un caractère lors du démarrage du système mais il n'envoyait rien après. En regardant le code, j'ai découvert que le UART ne retournait pas à son état IDLE où il est disponible pour envoyer des caractères. Par conséquent, le module à fait une seule passe de transmission (comportement assorti avec mes observations). La solution a été d'inclure la remise à l'état IDLE après la fin d'une transmission.

- *Bon envoi du caractère « B », mauvais envoi du caractère « e »*

Ce bug a été assez subtile et élusif. L'envoi du caractère « B » (0b01000010 en ASCII) a été réussi mais pas le caractère « e » (0b01101001 en ASCII). Au lieu du caractère « e », je voyais un caractère non-lisible (un caractère de contrôle du terminal). Comment est-ce que c'est possible qu'un caractère est envoyé sans problème mais un autre ne l'est pas ?

Le problème c'était que j'avais reversé l'ordre des bits lors de l'envoi de données. C'est-à-dire qu'au lieu d'envoyer le LSB du caractère ASCII en premier, j'envoyais le MSB ! Par conséquent, le caractère B est envoyé sans souci car la byte est en fait réversible (les bits forment un miroir au milieu). Quelle coïncidence. Le caractère « e » n'est pas réversible et ainsi son envoi ne fonctionnait pas.

Une fois j'ai modifié le code pour envoyer le LSB les 2 caractères pouvaient être envoyés sans problème.

Conclusion

Voici ce qui conclut la présentation du système implanté sur FPGA: Générateur des termes Fibonacci. Le système est capable de générer à la cadence de 1 Hz, un nombre de termes (spécifiés par l'utilisateur) qui appartiennent à la suite Fibonacci, puis les afficher sur des afficheurs 7 segments LEDs. Des termes erronées (une conséquence d'overflow au niveau du générateur) sont détectables et un message d'avertissement est envoyé à l'utilisateur via une communication UART. Une LED de status clignote à la même fréquence du fonctionnement du système. L'utilisateur peut « reset » le système et sélectionner le baud rate de la liaison série, UART (9600 baud ou 196200 baud). Des améliorations futures peuvent être apportées à ce système pour envoyer des caractères de contrôles (nouvelle ligne \n, retour à chariot \r, etc.) et/ou des messages plus longs et descriptifs pour l'utilisateur. Grâce à la nature paramétrable du système (utilisation des génériques), il est facilement extensible pour incorporer plus de digits pour les afficheurs.

Références

[1] «Double dabble algorithm,» [En ligne]. Available: https://en.wikipedia.org/wiki/double_dabble.

[2] J. Silver, «Électronique Numérique 1 - VHDL (M1): Générateur de nombres Fibonacci,» Strasbourg, 2020.

ELECTRONIQUE NUMERIQUE 1 – VHDL

MNE (M1)

Rapport TP N°3 : GENERATEUR DE NOMBRES - SUITE DE FIBONACCI

Écrit par : Justin Silver

Groupe B (TP)

Simulation : 07/12/2020
Rapport soumis : 11/12/2020

Fichier électronique : rapport_TP3_SILVER.pdf

1. Module VHDL

Entité du générateur de la suite de Fibonacci

Pour réaliser le compteur Fibonacci, l'entité a été construite selon le cahier de charges et les indications données dans l'énoncé du TP. Notamment, avec le vecteur d'entrée « nb_termes » et le générique « TAILLE_SORTIE », le générateur de termes Fibonacci est paramétrable.

```vhdl
library ieee;
use ieee.std_logic_1164.all;

-- "numeric_std" est utilisée au lieu de std_unsigned.all/std_signed.all
-- ces dernières cast automatiquement les vecteurs à signed/unsigned
-- pour faire les opérations arithmétiques
use ieee.numeric_std.all;

entity fibonacci is
  generic (
    -- ce terme détermine le nombre de bits pour le vecteur de sortie
    -- 'sequence_number'
    TAILLE_SORTIE : integer := 32
  );

  port (
    -- initialisation synchrone
    init : in std_logic;

    -- enable synchrone
    enable : in std_logic;

    -- à chaque front horloge, un nouveau terme de la suite Fibonacci est
    -- calculée si le signal 'enable' est actif
    clk : in std_logic;

    -- la valeur de ce vecteur d'entrée détermine
    -- le nombre de termes à compter de la suite Fibonacci
    nb_termes : in std_logic_vector(7 downto 0);

    -- le terme courant de la suite Fibonacci
    sequence_number : out std_logic_vector((TAILLE_SORTIE-1) downto 0) := (others => '0');

    -- tant que ce signal est actif, les valeurs que prend le vecteur
    -- de sortie 'sequence_number' sont éronnées
    overflow  : out std_logic
  );
end fibonacci;
```

Figure 1. Entité du compteur Fibonacci

Architecture du générateur de la suite de Fibonacci

Pour générer les bons termes de la suite de Fibonacci et pour gérer la détection d'overflow, 4 signaux intermédiaires sont inclus dans l'architecture. Le fonctionnement interne du générateur est le suivant : sur chaque front d'horloge, le terme précèdent prend la valeur du terme actuel, et le terme actuel prend la valeur du terme futur calculé au dernier cycle d'horloge.

```vhdl
architecture logic of fibonacci is
  -- signaux intérmediares
  signal terme_futur : std_logic_vector((TAILLE_SORTIE-1) downto 0);
  signal terme_actuel : std_logic_vector((TAILLE_SORTIE-1) downto 0);
  signal terme_precedent : std_logic_vector((TAILLE_SORTIE-1) downto 0);
  signal count_nb_termes : std_logic_vector(7 downto 0) := (others => '0');
begin
```

Figure 2. Zone déclarative de l'architecture du compteur Fibonacci

L'architecture de ce compteur est décomposée en plusieurs process. Le premier modifie seulement le terme actuel, le terme précèdent, et le nombre de termes calculés. Le deuxième gère l'état du terme futur. Une fois que le terme précèdent et le terme actuel sont mis à jour à la fin du premier process, un autre process dédié au calcul du terme futur process se réveille et s'exécute.

Nous remarquons plusieurs choses intéressantes dans le premier process.

- Si le signal d'initialisation OU le générateur a atteint le nombre de termes demandés, tout est remis à 0 sauf pour le compte du nombre de termes (ce qui est mis à 1). Ceci car le chiffre « 0 » est le premier terme de la suite de Fibonacci.

- Quand le terme futur atteint une valeur plus petite que le terme actuel, un overflow s'est forcément produit d'après la formule du calcul de la suite de Fibonacci : **terme(i) = terme(i) + terme(i-1)...** C'est pourquoi le signal « overflow » est mis à 1 dans ce cas. Si la taille du vecteur de sortie permet d'atteindre la valeur maximale des termes de Fibonacci pour le nombre demandé, alors le signal overflow ne s'activera jamais.

- Car le count_nb_termes signal est un std_logic_vector, il faut le caster en type *unsigned* pour faire l'arithmétique : + 1. Il est ensuite reconverti en std_logic_vector.

```
-- ce process ne touche que les signaux:
-- terme_precedent, terme_actuel, count_nb_termes
process (clk)
begin
  if (clk'event and clk = '1') then

    -- est-ce qu'on recommence le comptage de la suite de Fibonacci?
    if (init = '1' OR count_nb_termes = nb_termes) then
      terme_actuel <= (others => '0');
      terme_precedent <= (others => '0');
      overflow <= '0';

      -- remet à 1 et pas à 0, car 0 est un terme dans la suite de Fibonacci
      count_nb_termes <= std_logic_vector(to_unsigned(1, count_nb_termes'length));

    -- est-ce qu'on continue à générer le prochain nombre de la suite de Fibonacci?
    elsif enable = '1' then

      -- en régime normal, le terme_futur devrait toujours être >=
      -- le terme_actuel. sinon, le terme_futur a du faire un overflow
      if terme_futur < terme_actuel then
        overflow <= '1';
      end if;

      -- mise à jour des signaux intermediares
      terme_actuel <= terme_futur;
      terme_precedent <= terme_actuel;

      -- incrément le nombre de termes calculés
      count_nb_termes <= std_logic_vector(unsigned(count_nb_termes) + to_unsigned(1, count_nb_termes'length));

    end if;
  end if;
end process;

-- mettre à jour le vecteur de sortie
sequence_number <= terme_actuel;
```

Figure 3. Premier process du compteur Fibonacci

À la fin du premier process, nous mettons le vecteur de sortie « sequence_number » à jour avec le terme actuel ce qui vient de prendre le terme futur calculé au dernier cycle d'horloge.
Dans le process dédié au terme futur, le cas où une initialisation est demandée est prise en compte. Dans ce cas particulier, le terme actuel et le terme précèdent auraient du être mis à 0 par le premier process. Le terme futur passe à 1, car 1 est le deuxième terme dans la suite de Fibonacci.

Si une initialisation n'est pas demandée, alors le calcul du prochain terme dans la suite de Fibonacci est exécuté.

```vhdl
-- ce process ne touche que la valeur du terme_futur
process(terme_actuel, terme_precedent)

  -- ce constant est pour comparer quand les signaux intermediares = 0
  constant all_zeros : std_logic_vector((TAILLE_SORTIE-1) downto 0) := (others => '0');

begin
  -- quand terme_actuel = terme_precedent = 0, nous avons eu une initialisation
  -- OU le nb de termes demandés a été atteint
  if (terme_actuel = all_zeros AND terme_precedent = all_zeros) then
    terme_futur <= std_logic_vector(to_unsigned(1, terme_futur'length));
  else

    -- mettre a jour le terme futur avec des termes_actuel et terme_precedent
    -- qui viennent d'etre mise à jour
    terme_futur <= std_logic_vector(unsigned(terme_actuel) + unsigned(terme_precedent));

  end if;
end process;
```

Figure 4. Dernier process du compteur Fibonacci

2. Testbench/Simulation

Pour le testbench de ce module VHDL, nous expérimentons avec 2 valeurs différentes pour le générique : TAILLE_SORTIE. Une qui garde le compteur en régime normal et une autre qui force une erreur à se produire (overflow).

Dans le premier cas montré ci-dessous, nous remarquons le bon fonctionnement des signaux « enable » et « init ». Nous voyons également que le nombre de termes demandés est respecté, et une fois que ce nombre est atteint, un nouveau cycle commence.

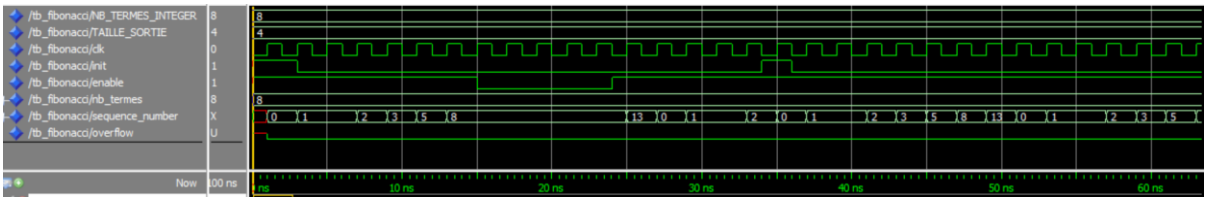

Figure 5. Simulation où le signal d'overflow n'est pas activé

Pour le deuxième cas, (Figure 6) l'utilisateur demande trop de nombres de la suite de Fibonacci qui peut produire le vecteur de sortie. Ceci est le cas car la sortie peut afficher jusqu'à la valeur 15 avec une TAILLE_SORTIE = 4, alors que l'utilisateur demande d'aller jusqu'à la valeur 55 (onzième terme dans la suite de Fibonacci). Une erreur est alors détectée par l'état de l'overflow. Il est important à remarquer que lorsqu'un nouveau cycle recommence, les premiers termes ne sont pas erronés et donc l'overflow n'est pas activé.

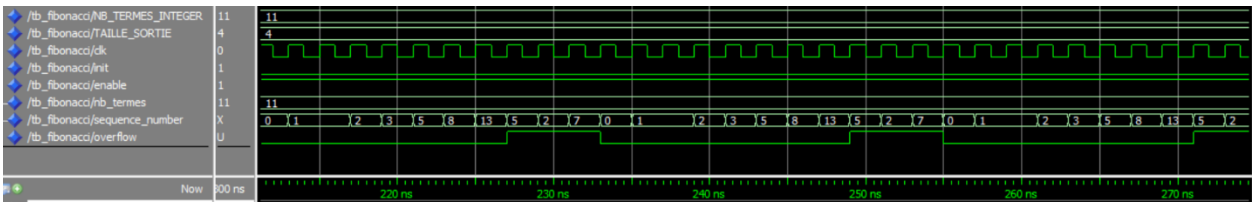

Figure 6. Simulation où le signal d'overflow s'active

Annexe 2. Module digit_types

```
library ieee;
use ieee.std_logic_1164.all;

package digit_types is
    -- unconstrained arraies (undefined length)
    -- index type range is defined when an object of these types is declared
    type bcd_digit is array (integer range <>) of std_logic_vector(3 downto 0);
    type ledseg_digit is array (integer range <>) of std_logic_vector(6 downto 0);
end package digit_types;
```

Annexe 3. Module baud_gen

```
library ieee;
use ieee.std_logic_1164.all;

entity baud_gen is
    generic (
        CLK_FREQ : integer := 50000000
    );

    port (
        baud_select : in std_logic;
        clk : in std_logic;
        rst : in std_logic := '0';

        baud_out : out std_logic := '0'
    );
end entity;

architecture logic of baud_gen is

    -- to determine the count for obtaining the specified baud rate, the following
    -- formula is used: CLK_FREQ/X = BAUD_RATE/2 => CLK_FREQ/(2*BAUD_RATE)

    -- example: to have a baud rate of 9600 (9600 bits/s) with a CLK_FREQ of 50 MHz,
    -- we have to count to 50*(10^6)/(19200), thus X ~= 2604. This means that every 2604 ticks,
    -- the baud_out signal's state will toggle. In one second, there will be 50 mil-
lion ticks, which means that
    -- the baud_out signal would have toggled ((50*10^6)/2600) times, ~= 19200.
    -- THUS, the frequency of the baud_out signal would be 19200/2, = 9600.

    constant COUNT_BAUD_9600 : integer := CLK_FREQ/(2 * 9600);
    constant COUNT_BAUD_19200 : integer := CLK_FREQ/(2 * 19200);

    constant BAUD_SELECT_9600 : std_logic := '0';
    constant BAUD_SELECT_19200 : std_logic := '1';

    signal baud_out_temp : std_logic := '0';
```

```
begin

    process (clk, rst)
        variable count : integer := 0;
    begin
        -- asynchronous reset
        if (rst = '1') then
            count := 0;
            baud_out_temp <= '0';

        elsif rising_edge(clk) then
            count := count + 1;
            -- if the user has selected a certain baud rate and the variable has reached the speci-
fied rate,
            -- then toggle baud_out_temp signal state. Thus we generate a clock at a fre-
quency of 9600 or 19200 Hz.
            -- NOTE: it's important to have an >= expression for the count, other-
wise the count might depass
            -- 9600 and/or 19200 if for example baud_select was high-z or any other signal type
            if (baud_select = BAUD_SELECT_9600 and count >= COUNT_BAUD_9600) or (baud_se-
lect = BAUD_SELECT_19200 and count >= COUNT_BAUD_19200) then
                count := 0;
                baud_out_temp <= not(baud_out_temp);
            end if;
        end if;
    end process;

    -- update the output whenever internal signal changes
    -- at the end of the process
    baud_out <= baud_out_temp;

end architecture;
```

Annexe 4. Testbench pour baud_gen

```
library ieee;
use ieee.std_logic_1164.all;

entity tb_baud_gen is
end tb_baud_gen;

architecture bench of tb_baud_gen is

    -- 50 MHz for visualizing both baud rates in ModelSim
    constant CLK_FREQ : integer := 50000000;

    signal baud_select : std_logic :='0';
    signal clk : std_logic := '0';
    signal rst : std_logic := '0';
    signal baud_out : std_logic;

    component baud_gen is
        generic (
            CLK_FREQ: integer
        );

        port (
            baud_select: in std_logic;
            clk: in std_logic;
            rst: in std_logic;
            baud_out: out std_logic
        );

    end component baud_gen;

begin

  inst_baud_gen : baud_gen
    generic map (
      CLK_FREQ => CLK_FREQ
    )

    port map (
        baud_select => baud_select,
        clk => clk,
        rst => rst,
        baud_out => baud_out
    );

    clk <= not(clk) after 10 ns;

    rst  <= '1',
            '0' after 15 ns;

    baud_select <= '0',
                   '1' after 1 ms;

end bench;
```

Annexe 5. Module bin2bcd

```
--------------------------------------------------------------------------------
-- source code adapted from: https://en.wikipedia.org/wiki/double_dabble
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.digit_types.bcd_digit;

entity bin2bcd is
    generic (
        -- input binary signal bit length
        BIN_IN_LENGTH : integer := 12;
        -- 4 digits means: ones, tens, hundreds, thousands
        NUM_BCD_DIGITS : integer := 4
    );
    port (
        bin_in : in std_logic_vector((BIN_IN_LENGTH - 1) downto 0);

        -- an array of N number of BCD digits each with 4 bits (see type package)
        bcd_out : out bcd_digit((NUM_BCD_DIGITS - 1) downto 0) := (others => (others => '0'))
    );
end bin2bcd;

architecture behavioral of bin2bcd is
begin
```

```
    process (bin_in)
        -- this holds the input binary number
        variable temp : std_logic_vector ((BIN_IN_LENGTH - 1) downto 0);

        -- each BCD digit takes up 4 bits, so we can easily determine the total bit width
        -- needed for this variable based on the generic
        variable bcd_temp : unsigned ((NUM_BCD_DIGITS * 4 - 1) downto 0) := (others => '0');
    begin
        -- zero the bcd variable
        bcd_temp := (others => '0');

        -- read input into temp variable
        temp((BIN_IN_LENGTH - 1) downto 0) := bin_in;

        -- following loop could be optimized since we do not need to check and
        -- add 3 for the first 3 iterations (number can never be > 4)
        for i in 0 to (BIN_IN_LENGTH - 1) loop

            -- loop through each BCD digit to check if result of the shifting number is grea-
ter than 4
            for j in 0 to (NUM_BCD_DIGITS - 1) loop

                -- algorithm to check each BCD digit going from LSB to MSB
                -- (3 downto 0, 7 downto 4, ... etc.)
                if bcd_temp((3 + 4 * j) downto j * 4) > 4 then
                    bcd_temp((3 + 4 * j) downto j * 4) := bcd_temp((3 + 4 * j) downto j * 4) + 3;
                end if;

            end loop;

            -- shift bcd_temp left by 1 bit, copy MSB of temp into LSB of bcd_temp
            bcd_temp := bcd_temp(NUM_BCD_DIGITS * 4 - 2 downto 0) & temp(BIN_IN_LENGTH - 1);

            -- shift temp left by 1 bit
            temp := temp((BIN_IN_LENGTH - 2) downto 0) & '0';

        end loop;

        -- set output after type conversion.
        -- again, the complicated looking formula for indexing the bcd_temp variable
        -- is to obtain the individual BCD digits and place them into the bcd_out array
        for i in 0 to (NUM_BCD_DIGITS - 1) loop
            bcd_out(i) <= std_logic_vector(bcd_temp((3 + 4 * i) downto i * 4));
        end loop;

    end process;

end behavioral;
```

Annexe 6. Testbench pour bin2bcd

```
library ieee;
use ieee.std_logic_1164.all;
use work.digit_types.bcd_digit;

entity tb_bin2bcd is
end tb_bin2bcd;

architecture behavior of tb_bin2bcd is
    component bin2bcd
        generic (
            BIN_IN_LENGTH : integer;
            NUM_BCD_DIGITS: integer
        );
        port (
            bin_in : in  std_logic_vector ((BIN_IN_LENGTH-1) downto 0);
            bcd_out: out bcd_digit((NUM_BCD_DIGITS-1) downto 0)
        );
    end component;

    constant BIN_IN_LENGTH : integer := 16;
    constant NUM_BCD_DIGITS: integer := 4;

    signal bin_in : std_logic_vector((BIN_IN_LENGTH-1) downto 0) := (others => '0');

    -- (can be omitted)
    signal clk : std_logic := '0';

    signal bcd_out : bcd_digit((NUM_BCD_DIGITS-1) downto 0) := (others => (others => '0'));

    -- clock period definitions
    -- (can be omitted)
    constant CLK_PERIOD : time := 10 ns;
```

```
begin
    -- instantiate the unit under test (uut)
    uut: bin2bcd
        generic map (
            BIN_IN_LENGTH => BIN_IN_LENGTH,
            NUM_BCD_DIGITS => NUM_BCD_DIGITS
        )
        port map (
            bin_in => bin_in,
            bcd_out => bcd_out
        );
   -- clock process definitions
   -- the whole process could be omitted (not a sequential design)
  clk_process : process
  begin
    clk <= '0';
    wait for CLK_PERIOD/2;
    clk <= '1';
    wait for CLK_PERIOD/2;
  end process;

   -- stimulus process
   stim_proc: process
   begin
      -- hold reset state for 30 ns.
      wait for 30 ns;

      wait for CLK_PERIOD*10;

      -- should return 4095
      bin_in <= x"0fff";
      wait for CLK_PERIOD*10;
      assert bcd_out(3) & bcd_out(2) & bcd_out(1) & bcd_out(0) = x"4095" severity error;

      -- should return 0
      bin_in <= x"0000";
      wait for CLK_PERIOD*10;
      assert bcd_out(3) & bcd_out(2) & bcd_out(1) & bcd_out(0) = x"0000" severity error;

      -- should return 2748
      bin_in <= x"0abc";
      wait for CLK_PERIOD*10;
      assert bcd_out(3) & bcd_out(2) & bcd_out(1) & bcd_out(0) = x"2748" severity error;

      -- should return 9999
      bin_in <= x"270F";
      wait for CLK_PERIOD*10;
      assert bcd_out(3) & bcd_out(2) & bcd_out(1) & bcd_out(0) = x"9999" severity error;

      -- should return 312
      bin_in <= x"0138";
      wait for CLK_PERIOD*10;
      assert bcd_out(3) & bcd_out(2) & bcd_out(1) & bcd_out(0) = x"0312" severity error;

      wait;
   end process;
end;
```

Annexe 7. Module blinky

```
library ieee;
use ieee.std_logic_1164.all;

entity blinky is
    generic (
        -- based on a 50 MHz onboard clock to have a 1 Hz LED blinky
        led_period : integer := 50000000/2
    );
    port (
        clk : in std_logic;
        rst : in std_logic := '0';
        led_out : out std_logic
    );
end entity;

architecture logic of blinky is
    signal led_temp : std_logic := '0';
begin
    process (clk)
        variable num_cycles : integer := 0;
    begin
        if rising_edge(clk) then
            if rst = '1' then
                num_cycles := 0;
                led_temp <= '0';

            elsif rst = '0' then
                if (num_cycles = led_period) then

                    -- count this clock edge as being part of led_period
                    num_cycles := 1;
                    led_temp <= not(led_temp);

                else
                    num_cycles := num_cycles + 1;
                end if;
            end if;
        end if;

    end process;

    -- update led_out
    led_out <= led_temp;

end architecture;
```

Annexe 8. Module char_select

```
library ieee;
use ieee.std_logic_1164.all;

-- simple mux to select which ASCII character to send via UART
-- The char_sel signal is controlled by the carry bit on the fibonacci gen.
-- If the cary is HIGH (overflow with the fibonacci generator) then we'll send an error char
-- Otherwise, we'll send an ok char

entity char_select is
    port (
        char_sel : in std_logic;
        char_out : out std_logic_vector(7 downto 0)
    );
end entity;

architecture logic of char_select is
begin
    with char_sel select
        char_out <= x"42" when '0', -- the ASCII character 'B' represents 'BON'
                    x"45" when '1', -- the ASCII character 'E' represents 'ERREUR'
                    x"00" when others; -- we could add other chars if needed
end architecture;
```

Annexe 9. Module fibonacci_gen

```
library ieee;
use ieee.std_logic_1164.all;

-- "numeric_std" est utilisée au lieu de std_unsigned/std_signed
-- ces dernières cast automatiquement les vecteurs à signed/unsigned
-- pour faire les opérations arithmétiques
use ieee.numeric_std.all;

entity fibonacci_gen is
    generic (
        -- ce terme détermine le nombre de bits pour le vecteur de sortie
        -- 'sequence_number'
        TAILLE_SORTIE : integer := 32
    );

    port (
        -- initialisation synchrone
        init : in std_logic;

        -- enable synchrone
        enable : in std_logic;

        -- à chaque front horloge, un nouveau terme de la suite Fibonacci est
        -- calculée si le signal 'enable' est actif
        clk : in std_logic;

        -- la valeur de ce vecteur d'entrée détermine
        -- le nombre de termes à compter de la suite Fibonacci
        nb_termes : in std_logic_vector(7 downto 0);

        -- le terme courant de la suite Fibonacci
        sequence_number : out std_logic_vector((TAILLE_SORTIE - 1) downto 0) := (others => '0');

        -- tant que ce signal est actif, les valeurs que prend le vecteur
        -- de sortie 'sequence_number' sont éronnées
        overflow : out std_logic
    );
end fibonacci_gen;

architecture logic of fibonacci_gen is
    -- signaux intérmediares
    signal terme_futur : std_logic_vector((TAILLE_SORTIE - 1) downto 0);
    signal terme_actuel : std_logic_vector((TAILLE_SORTIE - 1) downto 0);
    signal terme_precedent : std_logic_vector((TAILLE_SORTIE - 1) downto 0);
    signal count_nb_termes : std_logic_vector(7 downto 0) := (others => '0');
begin
```

```
    -- ce process ne touche que les signaux:
    -- terme_precedent, terme_actuel, count_nb_termes
    process (clk)
    begin
        if rising_edge(clk) then
            -- est-ce qu'on recommence le comptage de la suite de Fibonacci?
            if (init = '1' or count_nb_termes = nb_termes) then
                terme_actuel <= (others => '0');
                terme_precedent <= (others => '0');
                overflow <= '0';
                -- remet à 0 et pas à 1, meme si 0 est un terme dans la suite de Fibonacci
                -- je fais cela car sinon la valeur demande par le nb_termes en entree est toujours
                -- une unite plus grande que ce qui est compte par le sequence_number
                count_nb_termes <= (others => '0');

            -- est-ce qu'on continue à générer le prochain nombre de la suite de Fibonacci?
            elsif enable = '1' then

                -- en régime normal, le terme_futur devrait toujours être >=
                -- le terme_actuel. sinon, le terme_futur a du faire un overflow
                if terme_futur < terme_actuel then
                    overflow <= '1';
                end if;

                -- mise à jour des signaux intermediares
                terme_actuel <= terme_futur;
                terme_precedent <= terme_actuel;

                -- incrément le nombre de termes calculés
                count_nb_termes <= std_logic_vector(unsigned(count_nb_termes) + to_unsigned(1, count
_nb_termes'length));

            end if;
        end if;
    end process;

    -- mettre à jour le vecteur de sortie
    sequence_number <= terme_actuel;

    -- ce process ne touche que la valeur du terme_futur
    process (terme_actuel, terme_precedent)

        -- ce constant est pour comparer quand les signaux intermediares = 0
        constant all_zeros : std_logic_vector((TAILLE_SORTIE - 1) downto 0) := (others => '0');
    begin
        -- quand terme_actuel = terme_precedent = 0, nous avons eu une initialisation
        -- OU le nb de termes demandés a été atteint
        if (terme_actuel = all_zeros and terme_precedent = all_zeros) then
            terme_futur <= std_logic_vector(to_unsigned(1, terme_futur'length));
        else
            -- mettre a jour le terme futur avec des termes_actuel et terme_precedent
            -- qui viennent d'etre mise à jour
            terme_futur <= std_logic_vector(unsigned(terme_actuel) + unsigned(terme_precedent));

        end if;
    end process;
end logic;
```

Annexe 10. Testbench fibonacci_gen

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-- termes Fibonacci pour réference
-- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144

entity tb_fibonacci_gen is
end tb_fibonacci_gen;

architecture bench of tb_fibonacci_gen is

  -- avec une taille de 4, nous avons la possibilité d'aller
  -- jusqu'à 13 dans la suite de Fibonacci
  constant TAILLE_SORTIE : integer := 4;

  -- ************** pas d'overflow ***************
  -- l'utilisateur veut aller jusqu'au 13
  constant NB_TERMES_INTEGER : integer := 8;

  -- ************** overflow ***************
  -- l'utilisateur veut aller jusqu'au 55
  --constant NB_TERMES_INTEGER : integer := 11;

  signal init : std_logic;
  signal enable : std_logic;
  signal clk : std_logic := '0';
  signal nb_termes : std_logic_vector(7 downto 0);
  signal sequence_number : std_logic_vector((TAILLE_SORTIE-1) downto 0);
  signal overflow  : std_logic;

  component fibonacci_gen is
    generic (
      TAILLE_SORTIE : integer
    );

    port (
      init : in std_logic;
      enable : in std_logic;
      clk : in std_logic;
      nb_termes : in std_logic_vector(7 downto 0);
      sequence_number : out std_logic_vector((TAILLE_SORTIE-1) downto 0);
      overflow  : out std_logic
  );
  end component fibonacci_gen;
```

```
begin

  inst_fibonacci_gen : fibonacci_gen
    generic map (
      TAILLE_SORTIE => TAILLE_SORTIE
    )

    port map (
      init => init,
      enable => enable,
      clk => clk,
      nb_termes => nb_termes,
      sequence_number => sequence_number,
      overflow => overflow
    );

  clk <= not(clk) after 1 ns;

  enable  <= '1',
             '0' after 15 ns,
             '1' after 24 ns;


  init <= '1',
          '0' after 3 ns,
          '1' after 34 ns,
          '0' after 36 ns;

  -- definir combien de termes on veut de la suite fibonacci
  nb_termes <= std_logic_vector(to_unsigned(NB_TERMES_INTEGER, nb_termes'length));


end bench;
```

Annexe 11. Module edge_divider

```
library ieee;
use ieee.std_logic_1164.all;

entity edge_divider is
    generic (
        divide_edge_value : integer := 50
    );
    port (
        clk : in std_logic;
        rst : in std_logic := '0';
        impulse_out : out std_logic
    );
end edge_divider;
architecture logic of edge_divider is
begin
    process (clk)
        variable num_cycles : integer := 0;
    begin
        if rising_edge(clk) then
            if rst = '1' then
                num_cycles := 0;
                impulse_out <= '0';

            elsif rst = '0' then
                if (num_cycles = divide_edge_value) then

                    -- count this clock edge as being part of divide_edge_value
                    num_cycles := 1;
                    impulse_out <= '1';

                else
                    num_cycles := num_cycles + 1;
                    impulse_out <= '0';
                end if;
            end if;
        end if;
    end process;
end architecture;
```

Annexe 12. Testbench edge_divider

```
library ieee;
use ieee.std_logic_1164.all;

entity tb_edge_divider is
end tb_edge_divider;

architecture bench of tb_edge_divider is

    signal clk : std_logic := '0';
    signal rst : std_logic := '0';
    signal impulse_out : std_logic;

    component edge_divider is
        generic (
            divide_edge_value : integer
        );

        port (
            clk: in std_logic;
            rst: in std_logic;
            impulse_out: out std_logic
        );

    end component edge_divider;

    -- in this test environement, we should see a 10 MHz impulse
    constant divide_edge_value : integer := 5;

    -- 50 MHz clock
    constant CLK_PERIOD : time := 20 ns;
begin
    clk_process : process
    begin
        clk <= '0';
        wait for CLK_PERIOD/2;
        clk <= '1';
        wait for CLK_PERIOD/2;
    end process;


  inst_edge_divider : edge_divider
    generic map (
      divide_edge_value => divide_edge_value
    )

    port map (
        clk => clk,
        rst => rst,
        impulse_out => impulse_out
    );

    rst  <= '1',
            '0' after 15 ns;

end bench;
```

Annexe 13. Module ledseg_decoder

```
library ieee;
use ieee.std_logic_1164.all;

-- COMMON ANODE LED 7 SEGMENT DECODER (n digits)
-- Segments: G F E D C B A

entity ledseg_decoder is
    port (
        decimal_in : in std_logic_vector(3 downto 0);
        -- decimal point not included
        ledseg_out : out std_logic_vector(6 downto 0)
    );
end entity;

architecture logic of ledseg_decoder is
begin
    process (decimal_in)
    begin
        case (decimal_in) is
            when x"0" => ledseg_out <= b"1000000";
            when x"1" => ledseg_out <= b"1111001";
            when x"2" => ledseg_out <= b"0100100";
            when x"3" => ledseg_out <= b"0110000";
            when x"4" => ledseg_out <= b"0011001";
            when x"5" => ledseg_out <= b"0010010";
            when x"6" => ledseg_out <= b"0000010";
            when x"7" => ledseg_out <= b"1011000";
            when x"8" => ledseg_out <= b"0000000";
            when x"9" => ledseg_out <= b"0010000";

                -- invalid inputs turn off all segments
            when others => ledseg_out <= b"1111111";
        end case;
    end process;
end architecture;
```

Annexe 14. n_ledseg_decoder

```
library ieee;
use ieee.std_logic_1164.all;
use work.digit_types.ledseg_digit;
use work.digit_types.bcd_digit;

-- COMMON ANODE LED 7 SEGMENT DECODER (n digits)
-- Segments: G F E D C B A
entity n_ledseg_decoder is
    generic (
        NUM_DIGITS : integer := 4
    );
    port (
        decimal_in : in bcd_digit((NUM_DIGITS - 1) downto 0);
        ledseg_out : out ledseg_digit(NUM_DIGITS - 1 downto 0) := (others => (others => '1'))
    );
end entity;

architecture logic of n_ledseg_decoder is
    component ledseg_decoder is
        port (
            decimal_in : in std_logic_vector(3 downto 0);
            ledseg_out : out std_logic_vector(6 downto 0)
        );
    end component ledseg_decoder;
begin

    build_n_digits : for i in 0 to (NUM_DIGITS - 1) generate
        -- dont need to do specify ledseg_decoder architecture
        -- (there's only one)
    begin
        decoder : ledseg_decoder
        port map(
            decimal_in => decimal_in(i),
            ledseg_out => ledseg_out(i)
        );
    end generate build_n_digits;

end architecture;
```

Annexe 15. Module system_control

```
--============================================================
-- Controller block for the system
--============================================================
-- TASK 1:
-- takes in 1 Hz impulse, delays it by n # of clk cycles.
-- This assures propagation of overflow bit through Fibonacci
-- generator module.
------------------------------------------------------------
-- TASK 2 (NOT YET IMPLEMENTED):
-- send CR control character for PC serial terminal
------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity system_control is
    generic (
        clk_cycle_delay : integer := 5
    );
    port (
        clk : in std_logic;
        rst : in std_logic;

        -- impulse
        data_in : in std_logic;

        -- delayed impulse
        data_out : out std_logic := '0'
    );
end entity;

architecture logic of system_control is

    -- state machine
    type state_type is (IDLE, DELAY_TRANSMIT, BEGIN_TRANSMIT);

    -- register to hold the current state
    signal state : state_type;

begin
```

```
    process (clk, rst)
        variable count : integer := 0;
    begin
        -- asynchronous rst
        if rst = '1' then
            count := 0;
            state <= IDLE;
        elsif rising_edge(clk) then
            case state is
                when IDLE =>
                    -- pulse detection
                    if data_in = '1' then
                        -- we should start counting clock cycles since the state machine
                        -- has an intrinsic 1 clk cycle delay between states
                        count := count + 1;
                        state <= DELAY_TRANSMIT;
                    else
                        state <= IDLE;
                    end if;

                when DELAY_TRANSMIT =>
                    if count >= clk_cycle_delay then
                        count := 0;
                        state <= BEGIN_TRANSMIT;
                    else
                        count := count + 1;
                        state <= DELAY_TRANSMIT;
                    end if;

                when BEGIN_TRANSMIT =>
                    -- keep data_out active for several clock cycles
                    if count >= clk_cycle_delay then
                        count := 0;
                        state <= IDLE;
                    else
                        count := count + 1;
                        state <= BEGIN_TRANSMIT;
                    end if;
            end case;
        end if;
    end process;

    process (state)
    begin
        case state is
            when IDLE =>
                data_out <= '0';

            when DELAY_TRANSMIT =>
                data_out <= '0';

            when BEGIN_TRANSMIT =>
                data_out <= '1';
        end case;
    end process;
end architecture;
```

Annexe 16. Testbench system_control

```
library ieee;
use ieee.std_logic_1164.all;

entity tb_system_control is
end tb_system_control;

architecture behavior of tb_system_control is
    component system_control
        generic(
            clk_cycle_delay : integer
        );
        port(
            clk: in std_logic;
            rst: in std_logic;
            data_in: in std_logic;
            data_out : out std_logic
        );
    end component;
    constant clk_cycle_delay : integer := 5;
    signal rst : std_logic;
    signal data_in : std_logic;
    signal data_out : std_logic;
    signal clk : std_logic := '0';
    constant CLK_PERIOD : time := 20 ns;
begin
    uut: system_control
        generic map (
            clk_cycle_delay => clk_cycle_delay
        )
        port map (
            clk => clk,
            rst => rst,
            data_in => data_in,
            data_out => data_out
        );
    clk_process : process
    begin
        clk <= '0';
        wait for CLK_PERIOD/2;
        clk <= '1';
        wait for CLK_PERIOD/2;
    end process;
    -- stimulus process
    stim_proc: process
    begin
        rst  <= '1',
                '0' after 5 ns;
        -- simulation of impulse in
        data_in <= '0',
                   '1' after 10 ns,
                   '0' after 30 ns,
                   '1' after 470 ns,
                   '0' after 490 ns;
      wait;
   end process;
end;
```

Annexe 17. Module uart_control

```
--============================================================
-- Controller for UART module
--============================================================
-- TASK 1:
-- detect when to initiate a transmission from 50 MHz clock
-- domain (coming from 1 Hz input impulse)
------------------------------------------------------------
-- TASK 2:
-- implement a 'handshaking' protocol with uart_transmit
-- module in baud_clk domain to start data transmission.
--
-- 1) controller gets TX request (50 MHz clk domain)
-- 2) controller sends TX request to peripheral
-- 3) controller waits for TX request ACK (TX_DONE goes low)
-- 4) controller waits for TX_DONE = '1'
------------------------------------------------------------
-- TASK 3:
-- once transmit module declares that the byte transfer has
-- completed, uart_control block waits for 50 MHz clock
-- domain impulse to go low, before reinitializing the
-- UART and idling for a new data transmission request
------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity uart_control is
    port (
        -- tx request line from 50 MHz clock domain
        TX_REQ_IN : in std_logic;

        rst : in std_logic;
        clk : in std_logic;

        -- uart_transmit flag
        TX_DONE : in std_logic;

        -- send new tx request to uart_transmit
        TX_REQ_OUT : out std_logic := '0'
    );
end entity;

architecture logic of uart_control is

    -- state machine
    type state_type is (IDLE, WAIT_ACK, WAIT_EOT, WAIT_RESET);
    signal state : state_type;

begin
```

```
    process (clk, rst)
    begin
        -- asynchronous reset
        if rst = '1' then

            TX_REQ_OUT <= '0';
            state <= IDLE;

        elsif rising_edge(clk) then

            case state is

                -- wait for new TX request from 50 MHz domain
                when IDLE =>
                    if TX_REQ_IN = '1' then
                        -- request new TX (uart_transmit)
                        TX_REQ_OUT <= '1';
                        state <= WAIT_ACK;
                    else
                        TX_REQ_OUT <= '0';
                        state <= IDLE;
                    end if;

                -- wait for uart_transmit to acknowledge TX request
                -- by starting data transmission (TX_DONE goes low)
                when WAIT_ACK =>
                    if TX_DONE = '0' then
                        TX_REQ_OUT <= '0';
                        state <= WAIT_EOT;
                    else
                        TX_REQ_OUT <= '1';
                        state <= WAIT_ACK;
                    end if;

                -- wait for end of tranmission flag
                when WAIT_EOT =>
                    if TX_DONE = '1' then
                        state <= WAIT_RESET;
                    else
                        state <= WAIT_EOT;
                    end if;

                -- before sending new char, make sure TX request line in
                -- 50 MHz clk domain has gone low
                when WAIT_RESET =>
                    if TX_REQ_IN = '0' then
                        state <= IDLE;
                    else
                        state <= WAIT_RESET;
                    end if;
            end case;
        end if;
    end process;
end architecture;
```

Annexe 18. Testbench uart_control

```
library ieee;
use ieee.std_logic_1164.all;

entity tb_uart_control is
end tb_uart_control;

architecture bench of tb_uart_control is

    signal rst : std_logic := '0';
    signal clk : std_logic := '0';

    signal TX_REQ_IN : std_logic := '0';
    signal TX_DONE : std_logic := '1';

    signal TX_REQ_OUT : std_logic;

    component uart_control is
        port (
            rst : in std_logic;
            clk : in std_logic;
            TX_REQ_IN : in std_logic;
            TX_DONE : in std_logic;
            TX_REQ_OUT : out std_logic
        );

    end component uart_control;

begin

    inst_uart_control : uart_control
    port map (
        rst => rst,
        clk => clk,
        TX_REQ_IN => TX_REQ_IN,
        TX_DONE => TX_DONE,
        TX_REQ_OUT => TX_REQ_OUT
    );

    clk <= not(clk) after 10 ns;

    rst <= '1',
           '0' after 15 ns;

    TX_REQ_IN <= '1' after 25 ns,
                 '0' after 45 ns;

    -- simulating the uart_transmit block
    TX_DONE <= '0' after 100 ns,
               '1' after 180 ns;

end bench;
```

Annexe 19. Module uart_transmit

```
--============================================================
-- Tranmission/data transfer UART module
--============================================================
-- TASK 1:
-- detect a data transmission request
------------------------------------------------------------
-- TASK 2:
-- acknowledge TX request from UART controller (master)
-- by setting the TX_DONE flag low
------------------------------------------------------------
-- TASK 3:
-- 1) send start bit via TX serial line
-- 2) shift data byte through one bit at a time
-- 3) send stop bit and stop transmission
------------------------------------------------------------
-- TASK 4:
-- indicate to uart_control block that transmission is done
-- by setting TX_DONE flag high
------------------------------------------------------------
--_____________________________________________
-- transmission example with data: 0b11001010     |
--                                                |
-- CLK:  _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _  |
--        _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _   |
--                                                |
-- TX:    _____      _     _        _  _  _______  |
--             _  _     _     _  _                |
--       ^^^^ ^  ^  ^  ^  ^  ^  ^  ^  ^  ^  ^^^^  |
--      |IDLE|SA|D0|D1|D2|D3|D4|D5|D6|D7|SP|IDLE| |
--_____|_____|__|__|__|__|__|__|__|__|__|__|_____|
--

library ieee;
use ieee.std_logic_1164.all;

entity uart_transmit is
    generic (
        constant DATA_LENGTH : integer := 8
    );
    port (
        -- parallel data in
        data_in : in std_logic_vector((DATA_LENGTH - 1) downto 0);
        -- request line coming from uart_control block
        TX_REQ_IN : in std_logic;
        rst : in std_logic;
        -- clock coming from baud rate generator
        clk : in std_logic;
        --UART transmit data out
        --start bit is an active LOW
        UART_TX : out std_logic := '1';
        TX_DONE : out std_logic := '1'
    );
end entity;
architecture logic of uart_transmit is
    -- state machine
    type state_type is (IDLE, DATA_TX, END_TX);
    signal state : state_type;
    signal shift_register : std_logic_vector((DATA_LENGTH - 1) downto 0) := (others => '0');
begin
```

```
    process (clk, rst)
        variable bits_sent : integer := 0;
    begin
        -- asynchronous reset
        if rst = '1' then
            shift_register <= (others => '0');
            UART_TX <= '1';
            bits_sent := 0;
            TX_DONE <= '1';
            state <= IDLE;
        elsif rising_edge(clk) then
            case state is
                when IDLE =>
                    if TX_REQ_IN = '1' then
                        -- let uart_control know that a TX is underway
                        TX_DONE <= '0';
                        -- send start bit
                        UART_TX <= '0';
                        -- load shift register
                        shift_register <= data_in;
                        state <= DATA_TX;
                    else
                        -- not currently transmitting
                        UART_TX <= '1';
                        TX_DONE <= '1';
                        state <= IDLE;
                    end if;

                when DATA_TX =>
                    -- are we done sending the data byte?
                    if bits_sent >= DATA_LENGTH then
                        bits_sent := 0;
                        -- send STOP bit
                        UART_TX <= '1';
                        state <= END_TX;
                    else
                        -- serial out the LSB of the shift_register
                        UART_TX <= shift_register(0);
                        -- increment number of bits sent
                        bits_sent := bits_sent + 1;
                        -- shift bits
                        for i in 0 to (DATA_LENGTH - 2) loop
                            shift_register(i) <= shift_register(i + 1);
                        end loop;
                        state <= DATA_TX;
                    end if;

                -- an explicit state is included for the end of transmission
                -- so that the STOP bit has time to be sent
                when END_TX =>
                    TX_DONE <= '1';
                    state <= IDLE;

            end case;
        end if;
    end process;
end architecture;
```

Annexe 20. Testbench uart_transmit

```
library ieee;
use ieee.std_logic_1164.all;

entity tb_uart_transmit is
end tb_uart_transmit;

architecture bench of tb_uart_transmit is
    constant DATA_LENGTH : integer := 8;
    signal rst : std_logic := '0';
    signal clk : std_logic := '0';
    signal data_in: std_logic_vector((DATA_LENGTH-1) downto 0);
    signal TX_REQ_IN : std_logic :='0';
    signal TX_DONE : std_logic;
    signal UART_TX : std_logic;

    component uart_transmit is
      generic (
          constant DATA_LENGTH : integer := 8
      );
      port (
        rst: in std_logic;
        clk: in std_logic;
        data_in: in std_logic_vector((DATA_LENGTH-1) downto 0);
        TX_REQ_IN: in std_logic;
        TX_DONE: out std_logic;
        UART_TX: out std_logic
      );

    end component uart_transmit;

begin

  inst_uart_transmit : uart_transmit
    generic map (
      DATA_LENGTH => DATA_LENGTH
    )

    port map (
        clk => clk,
        rst => rst,
        TX_REQ_IN => TX_REQ_IN,
        data_in => data_in,
        TX_DONE => TX_DONE,
        UART_TX => UART_TX
    );

    clk <= not(clk) after 10 ns;
    rst  <= '1',
            '0' after 15 ns;
    data_in <= x"F0", -- 1111 0000
               x"00" after 90 ns;

    TX_REQ_IN <= '1' after 45 ns,
                 '0' after 95 ns;

end bench;
```