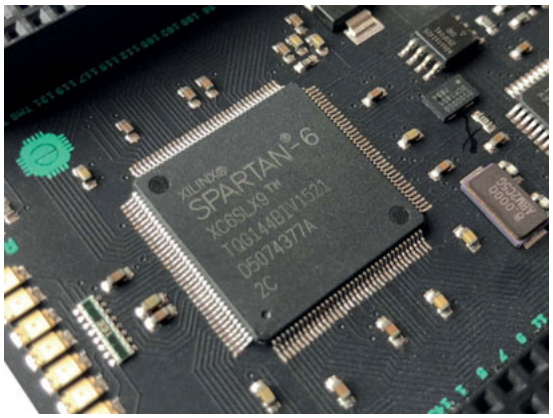
Arnaud Boudou
développeur chez ekito
co-fondateur du Fab Lab Sud31



À la découverte du FPGA

En tant que Maker ou simplement hobbyiste passionné, vous vous êtes peut-être lancé dans le développement autour des microcontrôleurs (Arduino, MSP43x, PIC, etc.). Et un jour, vous avez entendu parler des FPGA, sans forcément savoir de quoi il s'agissait exactement, à part peut-être de quelque chose comme les microcontrôleurs, mais en plus compliqué (mystérieux ?). Cet article est là pour vous aider à aborder ce nouveau continent.

Avant d'aborder plus en détails le FPGA, rappelons de quoi est généralement constitué un microcontrôleur :

- un processeur, qui exécute des instructions ;
- de la RAM, qui stockera les données temporaires liées à l'exécution du programme ;
- de la ROM, qui stockera le programme à destination du processeur ;
- des périphériques (ports séries, entrées / sorties numériques et / ou analogiques, etc.) pour communiquer avec le monde extérieur.

Toutes ces parties sont du point de vue physique des ensembles de composants (transistors, condensateurs, résistances…) gravés dans le silicone, à défaut de marbre, et définitivement figés. Pour ajouter de nouvelles fonctionnalités, vous devrez concevoir un nouveau circuit.

Le FPGA, qu'est-ce que c'est ?

À côté, nous avons le FPGA, qui pourrait être vu comme une feuille blanche. Sur cette feuille blanche, vous allez pouvoir dessiner à peu près ce que vous voulez. Un FPGA peut donc aussi être vu comme un composant électronique qui ne sait rien faire une fois sorti de sa boîte. Rentrons maintenant dans les détails. FPGA est un acronyme signifiant Field-Programmable Gate Array (soit en français : réseau de portes programmables in situ). C'est un circuit programmable (« programmable » devant être compris comme « configurable ») composé d'une grille d'éléments logiques simples. Programmer un FPGA signifie donc interconnecter ces éléments logiques pour y implémenter la fonctionnalité voulue. Cette fonctionnalité peut être : un additionneur, un clignoteur de LED, un circuit logique ou même un processeur. Et comme les éléments logiques de base ne sont normalement pas connectés entre eux, il est même possible d'implémenter plusieurs fonctionnalités indépendantes les unes des autres. Vous êtes seulement limité par vos compétences et le nombre d'éléments logiques de votre FPGA.

J'ai parlé juste avant d'éléments logiques. Un élément logique est en général composé :

- d'une table de correspondance (LUT – Look Up Table) à plusieurs entrées et une sortie, servant à implémenter une équation en logique booléenne;
- une bascule (flip-flop) permettant de stocker l'état de la sortie de la table de correspondance.

Les éléments logiques sont reliés entre eux par une matrice de routage. Outre le circuit de programmation de la matrice et des éléments logiques, un FPGA peut contenir des circuits spécialisés, comme des multiplicateurs (coûteux à implémenter en logique booléenne), ou des blocs de RAM. L'intégration de ces circuits permet de laisser plus d'éléments logiques à votre disposition.

Les éléments logiques et la matrice de routage étant constitués de cellules SRAM, cela signifie qu'en cas de coupure de l'alimentation, le programme sera perdu. Il est donc généralement associé à une puce FPGA, un circuit externe contenant en ROM la configuration FPGA qui sera copiée sur la puce lors de l'allumage de l'ensemble.

Le FPGA, quelle utilité ?

Maintenant que nous avons une idée plus claire de ce que sont les FPGA, intéressons-nous à leur utilité.

Tout d'abord, les FPGA sont utilisés comme systèmes de prototypage de circuits avant que ceux-ci soient fabriqués de manière industrielle pour commercialisation. Faire fabriquer des circuits en dur coûtant plusieurs milliers d'euros / dollars / autre devise, toute erreur lors des tests de conception peut rapidement tourner au cauchemar financier. Passer par l'étape FPGA, qui est librement reconfigurable, permet de valider la logique via de simples reprogrammations, même si les performances sont inférieures au circuit final. Une autre utilité des FPGA est de pouvoir obtenir des circuits conçus pour des domaines spécifiques. Plutôt que d'utiliser un processeur exécutant un programme implémentant la solution souhaitée, il est possible d'implémenter cette même fonctionnalité dans un circuit spécialisé qui sera plus performant qu'un processeur généraliste. Par exemple, il est possible d'implémenter des algorithmes de reconnaissance d'images sur des FPGA plutôt que de faire mouliner un processeur généraliste.

Dernière utilisation d'un FPGA à laquelle je pense, celle de composant « généraliste » au sein d'équipements électroniques grand public. J'ai ainsi croisé une puce FPGA en désossant une imprimante laser. L'utilisation d'une puce FPGA permet d'apporter à la machine des fonctionnalités basiques qui pourront être améliorées et / ou corrigées par simple reconfiguration de la puce. Dans ce cas, la nouvelle configuration peut être apportée par une mise à jour des pilotes ou du firmware de l'équipement.

En reprenant chacun des points ci-dessous, il est possible de faire le parallèle avec les solutions de minage du bitcoin.

Au début étaient les programmes de minage se servant du **CPU généraliste** d'un ordinateur. Puis, quand la difficulté est devenue trop importante, les mineurs sont passés minage sur **GPU** ; des processeurs moins généralistes que les CPU et spécialisés dans les traitements mathématiques et parallèles.

Puis, quand la difficulté est devenue trop importante, les mineurs sont passés au circuit FPGA spécialisé. Le circuit était donc dédié uniquement au minage de bitcoin avec la logique ad-hoc, et chaque fois que des améliorations dans les algorithmes étaient disponibles, le circuit était reconfiguré avec une nouvelle version du programme.

Puis, quand la difficulté est devenue trop importante, les mineurs sont passés au **circuit ASIC** (Application-specific integrated circuit - circuit intégré propre à une application) : après le prototype FPGA, la puce est gravée en dur et suivant la même logique programmée. Les circuits ASIC offrant des performances nettement plus importantes que les FPGA, le minage de bitcoin a pu se poursuivre en toute quiétude.

Développer sur FPGA, les solutions

Maintenant que vous avez décidé de vous lancer dans le développement sur FPGA, vous allez devoir faire plusieurs choix, les deux principaux étant :

- quelle marque de FPGA ?
- quel langage ?

Les marques les plus fréquemment rencontrées sont les suivantes : Xilinx, Intel (anciennement Altera) et Lattice. Chacune de ces marques proposera à la fois ses gammes de FPGA, avec leurs environnements de développement dédiés. Pour la suite, je ne vais pas trop m'étendre sur les différences entre chaque gamme, cela nous ferait sortir du cadre de cet article. 1

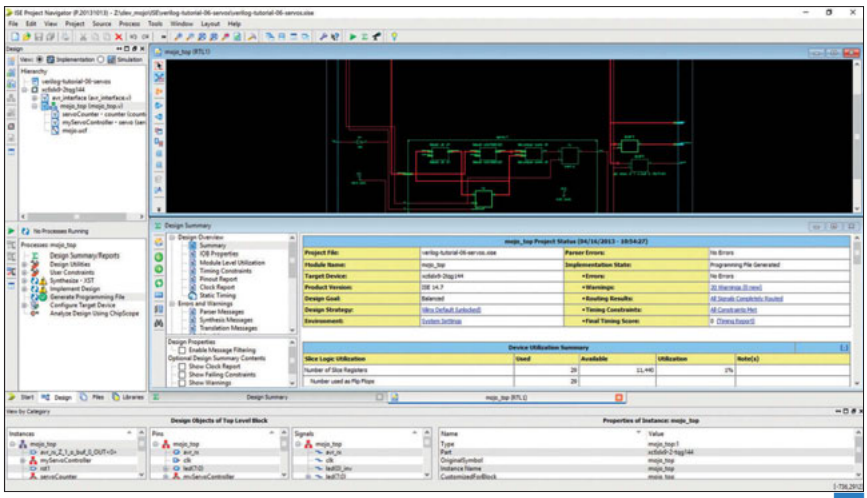
Xilinx

Cette marque propose principalement comme gamme de FPGA deux grandes familles : Spartan, et Virtex. En fonction de la famille visée, l'environnement de développement à utiliser n'est pas le même.

- ISE Design Suite (Windows / Linux) : cet environnement de développement cible la gamme Spartan-6 et Virtex-6 (ou versions précédentes). Il existe une version gratuite d'ISE, nommée « ISE Design Suite : WebPACK Edition », qui est plus limitée que la version normale, principalement sur des fonctions avancées. Gros défaut d'ISE, c'est que cette suite logicielle n'est plus maintenue par Xilinx, fonctionne mal sous Windows 10 (sauf bricolages pour l'installation) et pas du tout avec les distributions Linux utilisant le noyau 4.x.
- Vivado Design Suite (Windows / Linux) : cet environnement est cible la gamme Spartan-7 et Virtex-7. Là aussi, il existe une version gratuite nommée « Vivado HL WebPACK », elle aussi limitée en fonctionnalités. Contrairement à ISE, cet environnement de développement fonctionne parfaitement avec les versions récentes de Windows et Linux.

Intel

Plus précisément, Intel a racheté Altera en 2015, et a conservé son activité dans le domaine FPGA. La gamme de FPGA que vous êtes susceptible de rencontrer le plus fréquemment est la famille Cyclone. L'environnement de développement utilisé pour les produits Intel est Quartus (Windows / Linux). De la même manière que pour Xilinx, il existe une version gratuite limitée en fonctionnalités et en puces supportées. Mais elle suffira amplement pour se lancer dans le développement FPGA.



1

Lattice

Lattice est principalement connu chez les hobbyistes pour sa gamme de FPGA à faible coût, nommée iCE. L'environnement de développement indiqué en est Lattice Diamond (Windows / Linux). Il est possible d'obtenir une licence gratuite, qui sera limitée en produits supportés.

Il est à noter qu'un projet de développement d'une chaîne de compilation libre pour la gamme iCE40 existe. Cette chaîne de compilation vise à supporter Windows, Linux mais aussi macOS.

Vous noterez à la lecture des paragraphe précédents que, si vous préférez travailler sous macOS, comme c'est mon cas, vous allez devoir dégainer une machine virtuelle pour vous lancer dans le développement FPGA.

Une fois votre famille de prédilection choisie, il vous faut maintenant trouver une carte de développement. Pour vos débuts, pas besoin de vider le compte en banque, vous pouvez trouver le nécessaire pour une centaine d'euros. Honnêtement, il est assez difficile de conseiller une carte en particulier. Une liste plutôt exhaustive est disponible ici : https://joelw.id.au/FPGA/CheapFPGADevelopmentBoards.

Si vous souhaitez utiliser des FPGA Xilinx, j'aurais quand même tendance à vous déconseiller les cartes à base de Spartan-6 ou Spartan-3, qui vous obligeront à utiliser ISE (sauf bien entendu si vous avez la possibilité d'avoir un ordinateur un peu ancien sous le coude). Autre point à prendre en compte : la connexion à l'ordinateur de développement. Certaines cartes fournissent un connecteur USB qui sera directement reconnu par l'environnement de développement, d'autres auront un connecteur USB qui nécessitera un utilitaire dédié (fourni par le fabricant de la carte) pour le transfert du programme. Enfin certaines cartes ne fourniront qu'un connecteur JTAG, vous laissant la responsabilité de trouver le programmeur adéquat. Vous avez maintenant choisi votre famille de FPGA, votre carte de développement. Il ne vous reste plus qu'à

choisir le langage que vous allez utiliser. Les langages utilisés pour le développement FPGA sont appelés « HDL », pour « Hardware Description Langage ». En effet, quand on développe pour du FPGA, on n'écrit pas une liste d'instructions qui vont se dérouler les unes après les autres. À la place, on écrit la description d'un circuit au niveau logique. Cette description sera ensuite synthétisée en une série de portes logiques. Toutes les « instructions » s'exécuteront en parallèle et seront synchronisées sur la fréquence d'horloge du circuit.

Les deux langages majoritairement utilisés sont Verilog et VHDL. Le choix d'un langage est une simple question de préférence personnelle, ou même du langage utilisé pour les tutoriaux de votre carte de développement. Pour la petite histoire, VHDL est principalement utilisé en Europe, et Verilog de l'autre côté de l'Atlantique.

Tarifs

- Xilinx Vivado : de 2995 $ à 4295 $ en fonction de l'édition et du type de licence (fixe ou flottante)
- Xilinx ISE : une licence Vivado donne aussi droit à une licence ISE. Il n'est pas possible d'obtenir de licence ISE séparément.
- Intel Quartus : de 2995 $ à 4995 $ en fonction de l'édition et du type de licence

Premier programme sur FPGA

Il est maintenant temps d'écrire vos premiers programmes pour FPGA. Les exemples donnés sont pour la carte de développement Mojo v3 (https://embeddedmicro.com/products/mojo-v3.html), équipée d'une puce Spartan-6, et donc nécessitant Xilinx ISE, et seront écrits en Verilog. Ces exemples de code devraient être relativement aisés à adapter à une autre carte de développement. À cette puce Spartan est associé un microcontrôleur AVR faisant le lien avec l'ordinateur pour la programmation de la puce FPGA, et stockant le programme afin de pouvoir le réinstaller en cas de coupure de l'alimentation. Je partirai du principe que le projet que vous initialiserez dans votre environnement de développement sera correctement configuré pour votre carte. Il y a trop de combinaisons possibles pour que j'aborde ce sujet, je vous conseille donc de vous tourner vers la documentation de votre constructeur.

Vous le savez sûrement, mais le « hello world » du développement mettant en jeu de l'électronique est le clignoteur de LED : le résultat est facilement visible, et facilement modifiable. Je ne vais donc pas déroger à la tradition. En revanche, je vais commencer par le clignoteur de LED manuel …

Le clignoteur de LED manuel

La carte Mojo V3 possède entre autres 8 LEDs côte à côte et un bouton reset. Le but du jeu est de faire s'allumer une des LEDs lors de l'appui sur le bouton.

Lors de la création de votre projet, un fichier « mojo_top.v » est créé. Celui-ci représente le module dit « top level », qui implémentera d'éventuels sous-modules, et représentera l'état final du circuit. Un sous-module représente quant à lui une fonctionnalité de votre circuit (un compteur, un additionneur, etc.).

L'extension « .v » correspond à un fichier de langage Verilog. Chaque éventuel sous-module doit être dans son propre fichier pour des questions de lisibilité et maintenance.

Un module commence par la ligne :

```
module nom_module(
 les_parametres_du_module
) ;
```

Et se termine par la ligne :

```
endmodule
```

Notre module « mojo_top » commence par les lignes suivantes :

```
module mojo_top(
  // Entrée horloge à 50 MHz
  input clk,
  // Entrée reliée au bouton reset (active à l'état bas)
  input rst_n,
  // Entrée venant de l'AVR de programmation, état haut quand AVR prêt
  input cclk,
  // Sortie vers les 8 LEDs embarquées
  output[7:0]led,
  // Connexions vers le port SPI de l'AVR
  output spi_miso,
  input spi_ss,
  input spi_mosi,
  input spi_sck,
  // Sélection du convertisseur analogique vers numérique de l'AVR
  output [3:0] spi_channel,
  // Connexions séries vers l'AVR
  input avr_tx, // AVR Tx => FPGA Rx
  output avr_rx, // AVR Rx => FPGA Tx
  input avr_rx_busy // AVR Rx buffer full
 );
```

Cette déclaration indique l'ensemble des signaux qui entrent et qui sortent du module. Vous remarquerez pour l'entrée « reset » un suffixe « _n ». C'est une convention de nommage indiquant que cette entrée est active à l'état bas.

La sortie « led » est sous la forme « output[7:0] ». Cela signifie qu'il s'agit de 8 connexions en sortie, indexées de 7 à 0. Rien n'empêche d'écrire « output[10:3] » ou « output[8:1] », ou même « output[0:7] », mais le plus sûr est de coller à la convention, à moins d'avoir de très bonnes raisons de s'en écarter.

Après la déclaration du module, nous allons ajouter une connexion venant du bouton reset, mais inversant sa valeur. Ce qui signifie qu'elle sera active à l'état haut.

Il faut donc rajouter la ligne suivante :

```
wire rst = ~rst_n;
```

Le tilde devant « rst_n » signifie qu'on inverse sa valeur.

Les quatre lignes suivantes du projet sont :

```
assign spi_miso = 1'bz;
assign avr_rx = 1'bz;
assign spi_channel = 4'bzzzz;

assign led = 8'b0;
```

Le mot clé « assign » permet de donner une valeur à une connexion

précédemment déclarée. La valeur est sous la forme suivante : la largeur en bit de la valeur, une apostrophe de séparation, le format de la valeur (« b » pour bit, « h » pour hexadécimal, « d » pour décimal), et la valeur proprement dite.
Dans le cas des valeurs en bits, on a quatre-sous valeurs possibles : « 0 », « 1 » (jusque-là, normal), « z » (qui signifie état de haute impédance, ou déconnecté) et « x » (qui signifie valeur inconnue, ou on ne s'en préoccupe pas).
Du coup, on peut lire les choses suivantes :

- « spi_miso » se voit affecter une valeur de 1 bit, égale à « z » ;
- « avr_rx » de même ;
- « spi_channel » se voit affecter une valeur de 4 bits, égale à « zzzz » ;
- « led » se voit affecter une valeur de 8 bits, égale à « 0 » (ou plus précisément « 00000000 »).

Au final, les trois premières connexions sont déconnectées par sécurité (elles relient l'AVR de programmation à la puce FPGA), et les 8 LEDs sont éteintes.
Nous allons maintenant remplacer la dernière ligne par les deux lignes suivantes :

```
assign led[6:0] = 7'b0;
assign led[7] = rst;
```

Dans ce cas, on forcera les LEDs 0 à 6 à être éteintes (valeur 0), et la LED 7 sera branchée sur le signal inversé venant du bouton reset. On termine enfin le module par la ligne

```
endmodule
```

Le code complet (nettoyé des commentaires) est donc le suivant :

```
module mojo_top(
  input clk,
  input rst_n,
  input cclk,
  output[7:0]led,
  output spi_miso,
  input spi_ss,
  input spi_mosi,
  input spi_sck,
  output [3:0] spi_channel,
  input avr_tx,
  output avr_rx,
  input avr_rx_busy
 );

 wire rst = ~rst_n;

 assign spi_miso = 1'bz;
 assign avr_rx = 1'bz;
 assign spi_channel = 4'bzzzz;

 assign led[6:0] = 7'b0;
 assign led[7] = rst;

endmodule
```

Une fois le programme envoyé sur la carte, vous pourrez allumer la LED 7 en appuyant sur le bouton reset, et l'éteindre en relâchant le bouton. Dans ce programme, il n'y a pas de notion de boucle ou autre, nous avons simplement créé des connexions entre le bouton reset et la LED 7, avec un inverseur de signal entre les deux. Vous pouvez voir sur l'image ci-dessous le résultat de ce qui va être implémenté sur la puce FPGA. De gauche à droite, vous avez le signal d'entrée « rst_n », le passage par l'inverseur, et la sortie vers les LEDs (le détail sur la LED exactement branchée à l'inverseur n'apparaît pas). 2
Appuyer sur un bouton pour faire clignoter une LED, c'est fatigant. Il serait donc bien de rendre automatique ce clignotement. Nous allons donc passer au clignoteur automatique.

Clignoteur automatique

Pour cela, nous allons créer un nouveau sous-module, nommé « blinker », dans un fichier justement nommé… « blinker.v » (bravo à ceux qui n'ont pas encore décroché).
L'entête du module va prendre la forme suivante :

```
module blinker(
  input clock,
  input reset,
  output blink
 );
```

Ce module aura en entrée le signal d'horloge (celle qui anime notre puce, ici cadencée à 50 MHz), le signal du bouton reset, et aura comme sortie l'état du clignotement (éteint / allumé)
Nous allons déclarer dans ce module deux registres de 25 bits.

```
reg [24:0] counter_d, counter_q;
```

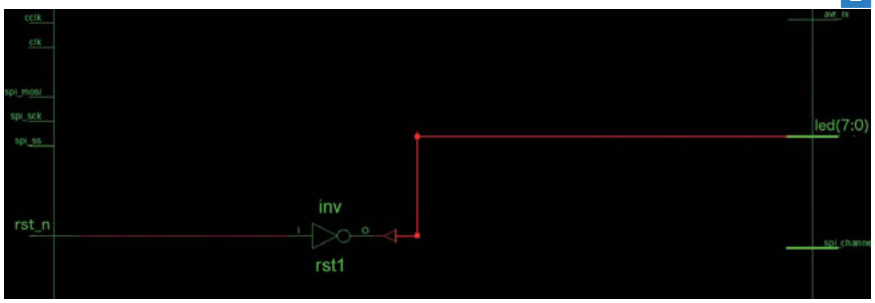
Ces registres seront implémentés sous la forme d'une chaine de 25 bascules de type D. Une bascule de type D est une mémoire de 1 bit avec une entrée D et une sortie Q. À chaque front d'horloge montant, la bascule va copier sur Q le signal présent sur D. Entre deux fronts montants d'horloge, la valeur de Q est conservée.
Nous décidons ensuite que la sortie « blink » du module sera reliée au 25ème bit du registre « counter_q ».

```
assign blink = counter_q[24];
```

À chaque fois que « counter_q » sera modifié, « counter_d » devra recevoir la valeur de « counter_q » incrémentée de 1 bit.

```
always @(counter_q) begin
  counter_d = counter_q + 1'b1;
end
```

2



À chaque front d'horloge, nous allons copier la valeur de « counter_d » dans « counter_q », déclenchant le bloc précédent. De plus, tant que le bouton reset sera maintenu appuyé, le compteur « counter_q » sera remis à 0.

```
always @(posedge clock) begin
 if (reset) begin
  counter_q <= 25'b0;
 end else begin
  counter_q <= counter_d;
 end
end
```

Et nous terminons notre sous-module par la ligne suivante :

```
endmodule
```

Le code complet du module est donc le suivant :

```
module blinker(
  input clock,
  input reset,
  output blink
 );

 reg [24:0] counter_d, counter_q;

 assign blink = counter_q[24];

 always @(counter_q) begin
  counter_d = counter_q + 1'b1;
 end

 always @(posedge clock) begin
  if (reset) begin
   counter_q <= 25'b0;
  end else begin
   counter_q <= counter_d;
  end
 end

endmodule
```

Vous aurez peut-être remarqué que l'assignation de valeur de « counter_q » se fait avec l'opérateur « <= » et non « = ». Cela signifie que cette assignation est non bloquante. Le résultat est que toutes les valeurs assignées avec cet opérateur au sein d'un même bloc sont assignées en parallèle et non pas séquentiellement.
Grosso modo, ce compteur va incrémenter un registre de 25 bits à chaque cycle d'horloge. Pendant la première moitié du temps, le bit 24 (le plus à gauche) sera à zéro (donc des valeurs 00000000000000000000000000 à 01111111111111111111111111), la sortie blink en fera de même. Et pendant l'autre moitié du temps, (donc des valeurs 10000000000000000000000000 à 11111111111111111111111111), le bit 24 sera à un, et de même pour la sortie blink. On aura donc 16 777 216 cycles d'horloge à 0 et autant à 1, ce qui fait pour une horloge à 50 MHz environ 0,34s à 1 et 0,34s à 0.
Il nous faut maintenant modifier le code du module « mojo_top.v » de la manière suivante. Il nous faut remplacer les lignes :

```
assign led[6:0] = 7'b0;
assign led[7] = rst;
```

Par les lignes suivantes :

```
assign led[7:1] = 7'b0;

blinker awesome_blinker (
 .clock(clk),
 .reset(rst),
 .blink(led[0])
);
```

Ces lignes de code implémentent un module « blinker » sous le nom « awesome_blinker », branchent les entrées du sous-module (ce qui commence par un point) avec les lignes d'horloge et de reset du module principal, et branchent la sortie du sous-module à la LED numéro 0.
Le code complet du module « mojo_top » est donc le suivant :

```
module mojo_top(
  input clk,
  input rst_n,
  input cclk,
  output[7:0]led,
  output spi_miso,
  input spi_ss,
  input spi_mosi,
  input spi_sck,
  output [3:0] spi_channel,
  input avr_tx,
  output avr_rx,
  input avr_rx_busy
 );

 wire rst = ~rst_n;

 assign spi_miso = 1'bz;
 assign avr_rx = 1'bz;
 assign spi_channel = 4'bzzzz;

 assign led[7:1] = 7'b0;

 blinker awesome_blinker (
  .clock(clk),
  .reset(rst),
  .blink(led[0])
 );

endmodule
```

A vous de coder !

Voilà, ce dernier bout de code conclut cet article sur la découverte de la programmation pour FPGA. J'espère que cet aperçu vous a donné envie d'aller plus loin dans ce domaine, qui sort complètement de l'ordinaire pour un développeur logiciel. •