Programmation d'un FPGA

Figure 1 : Carte de développement FPGA d'Alchitry Au.

Resources matérielles utiles

Si vous recherchez les produits utilisés dans cet article, vous les trouverez chez Elektor et chez SparkFun :

Alchitry Au FPGA Development Board

www.elektormagazine.fr/esfe-en-fpga1

Justin Rajewski (Alchitry)

Dans ce tutoriel, nous aborderons les notions de base des étapes de création de circuits FPGA et les blocs élémentaires à utiliser. De prime abord, un FPGA (*Field-Programmable Gate Array* = réseau d'opérateurs logiques programmables in situ) c'est assez intimidant pour un débutant, mais armé de logiciels accessibles, de matériel polyvalent et de quelques exemples judicieux, vous vous en sortirez sans peine.

Avant de nous lancer, rappelons qu'un FPGA ne se programme pas. [1] C'est un abus de langage qui s'explique par la similitude de la démarche : on écrit un texte qui, une fois transformé en binaire, est chargé dans le FPGA. Vous n'écrivez pas un programme, vous créez un circuit. Un langage de programmation ne peut pas servir à créer des circuits ; il s'agit en réalité d'un langage de description de circuits (*hardware description langage* ou HDL). [2]

Il serait fastidieux de dessiner le schéma d'une architecture complexe, et nous nous contentons de décrire le comportement attendu du circuit, ce sont ensuite les outils qui se chargent de trouver une solution concrète. Lors de la conception d'un projet FPGA, il faut garder à l'esprit que nous décrivons le matériel et que tout ce que nous écrivons sera transposé en un circuit physique. Il est possible de décrire des circuits impossibles à réaliser ou de décrire quelque chose qui semble simple, mais dont la réalisation nécessite une énorme quantité de ressources. C'est pourquoi il est essentiel de savoir comment pourra être réalisé le circuit que nous essayons de décrire.
Pour suivre ce tutoriel, il suffit d'une carte FPGA Alchitry Au (**fig. 1**) et d'un câble USB A-C réversible.

Structure d'un projet

En général, un HDL s'articule autour de l'idée de module. Un module est un bloc de circuit doté d'un certain nombre d'entrées et de sorties et contenant une logique pour les relier entre elles. À l'instar d'un programme que l'on décompose en fonctions, un module peut contenir des sous-modules. Il peut aussi être autonome.
Bien qu'il soit possible de regrouper un projet entier dans un seul module, il est préférable d'utiliser des modules plus petits pour réaliser chaque élément du projet. En décomposant notre projet en modules,

nous simplifions chaque partie sur laquelle nous travaillons à un moment donné. Certains modules seront conçus pour effectuer des tâches communes et seront utilisés à maintes reprises.
Lorsque nous débutons la conception, il est souvent utile de dessiner un schéma fonctionnel montrant les différents modules et leur interconnexion. Cela permet de définir l'étendue de notre projet et de le décomposer logiquement.

Lucid

Tout au long de ce tutoriel, nous utiliserons Lucid [3]. Lucid est un HDL conçu spécifiquement pour les FPGA. Il évite beaucoup de pièges (et croyez-moi, ils sont nombreux) dans lesquels il est facile de tomber avec d'autres HDL comme Verilog et VHDL.
Lucid est un outil extraordinaire pour débuter avec les FPGA. Souvent des personnes craignant des limitations avec Lucid ou voulant simplement passer à Verilog ou VHDL pour une autre raison me contactent. Si vous voulez bien débuter, faites-le avec Lucid. Il vous apprendra les bases de la conception sur matériel FPGA avant que vous n'exploriez d'autres HDL plus lourds. Si vous souhaitez le faire par la suite, ce n'est pas très difficile. Lucid est basé en gros sur Verilog et Alchitry Labs convertira Lucid en Verilog pour vous si vous voulez utiliser ailleurs vos modules hypersophistiqués.
Plongeons dans les entrailles d'un module.

Anatomie d'un module

Lorsque nous créons un projet, nous obtenons un module de niveau supérieur (= top level). C'est le module dont les entrées et sorties sont des entrées et sorties réelles sur les broches du FPGA. Pour tout projet Alchitry, il s'agit soit de *cu_top.luc*, soit de *au_top.luc* selon que nous utilisons la carte Cu ou Au. Les modules de niveau supérieur initiaux sont pratiquement identiques pour les deux cartes (**listage 1**).
La section de tête du module contient la déclaration des ports. C'est là que les entrées et sorties du module sont déclarées. Dans ce cas, comme il s'agit du module *top level*, il s'agit de signaux présents sur la carte elle-même (**listage 2**).
Vous avez peut-être remarqué que l'entrée *led est* suivie d'un numéro entre crochets. Il ne s'agit donc pas d'une, mais de huit entrées distinctes rassemblées en un tableau. Nous reviendrons sur la syntaxe des tableaux.
Un module peut comporter une liste de paramètres de personnalisation. Elle est omise ici, car inutile sur un module *top level* puisque les paramètres sont passés par le module parent qui l'instancie. Le terme d'*instanciation* est utilisé pour désigner le moment où un module (ou une autre ressource) est ajouté(e) à un projet. Il signifie qu'une *instance* de ce module (ou de cette autre ressource) est créée.
Dans un programme, l'appel à une fonction lance un code exécutable existant en un seul exemplaire, indépendamment du nombre de fois que la fonction est appelée. Au contraire, dans un projet FPGA, à chaque instanciation d'un module, l'ensemble du circuit qui le compose est dupliqué. Si, en tant que concepteurs, nous souhaitons réutiliser les mêmes ressources pour plusieurs tâches, il nous appartient de trouver comment jongler avec cela. À propos de l'instanciation, notons que nous instancions généralement **tout** ce dont le module a besoin **juste après** la déclaration de port.
La première ligne consiste à déclarer un signal en utilisant le mot-clé *sig* :

```
  sig rst;                    // reset signal
```

Les signaux ne sont pas des valeurs stockées en mémoire. Il faut les considérer comme des fils. Un fil peut avoir une valeur, ce n'est en fait qu'une connexion d'un point à un autre.

Listage 1.

```
module au_top (
   input clk,      // 100MHz clock
   input rst_n,    // reset button (active low)
   output led [8], // 8 user controllable LEDs
   input usb_rx,   // USB->Serial input
   output usb_tx   // USB->Serial output
  ) {

  sig rst;          // reset signal

 .clk(clk) {
   // The reset conditioner is used to
   // synchronize the reset signal to
   // the FPGA clock. This ensures the
   // entire FPGA comes out of reset at
   // the same time.
   reset_conditioner reset_cond;
  }

  always {
   reset_cond.in = ~rst_n; // input raw inverted
                           // reset signal
   rst = reset_cond.out;   // conditioned reset
    led = 8h00;            // turn LEDs off
    usb_tx = usb_rx;       // echo the serial data
  }
}
```

Listage 2.

```
    input clk,          // 100MHz clock
    input rst_n,        // reset button (active low)
    output led [8],     // 8 user controllable LEDs
    input usb_rx,       // USB->Serial input
    output usb_tx       // USB->Serial output
```

Dans ce projet, nous utilisons le signal *rst* comme paramètre de sortie du module *reset_conditioner.* Les deux lignes ci-dessous réalisent l'instanciation de ce module :

```
.clk(clk) {
    // The reset conditioner is used to synchronize
// the reset signal to the FPGA clock. This
// ensures the entire FPGA comes out of reset
// at the same time.
    reset_conditioner reset_cond;

  }
```

Pour instancier un élément, il suffit d'utiliser le nom de la ressource suivi du nom de cette instance particulière. Ainsi, la ligne `reset_conditioner reset_cond;` crée une instance du module *reset_conditioner* nommée *reset_cond*. Le bloc dans lequel cette instanciation est enveloppée s'appelle bloc de connexion. Il permet de connecter une entrée ou un paramètre (ayant un nom donné) de plusieurs modules au même signal.

Dans notre cas, nous connectons l'entrée *clk* au signal *clk* (qui est une entrée de notre module). La syntaxe est *.port(signal)* où *port est le* nom de l'entrée sur le module en cours d'instanciation et *signal* est le signal à y connecter.
Le module *reset_conditioner* a une entrée nommée *clk,* cette entrée est donc directement reliée au signal *clk de* notre module. Nous pouvons également la relier directement au module sur la ligne d'instanciation comme ceci :
reset_conditioner reset_cond(.clk(clk));
Nous ne le faisons pas ici parce que l'entrée *clk* fait partie de presque tous les modules et qu'il est pratique d'avoir un bloc comme celui-ci pour pouvoir simplement ajouter les autres instanciations qui doivent être connectées à *clk*.
Presque tous les modules disposent d'une entrée *clk* et souvent d'une entrée *rst* pour la réinitialisation. Nous verrons plus tard à quoi servent exactement l'horloge et l'initialisation. Nous pouvons ajouter plusieurs connexions au même bloc de connexion :
.clk(clk), .rst(rst) { ... }
Nous pouvons également imbriquer les blocs et comme tous les blocs qui utilisent *clk* n'utilisent pas *rst - en* général, nous verrons quelque chose de ce genre au début d'un module :

```
.clk(clk) {
    // clk only instantiations
    .rst(rst) {
        // rst and clk instantiations
    }

}
```

Les blocs *always*

C'est l'essence même du module : les blocs *always* sont les endroits où nous décrivons toute la logique qui s'y exprime. Ils renferment ce qu'on appelle de la logique combinatoire. Tout circuit numérique dont la sortie est exclusivement fonction de ses entrées est de la logique combinatoire. Il n'a ni état ni mémoire internes. Un additionneur est un bon exemple de logique combinatoire. La sortie est exclusivement déterminée par les deux nombres présents en entrée. Ni les chiffres entrés auparavant, ni le nombre de changements antérieurs n'ont d'importance. La sortie est toujours fonction des entrées en cours.
À l'intérieur du bloc "*always*", nous écrivons des instructions. Il y en a quatre types principaux :

- l'affectation ;
- l'instruction *if* (instruction conditionnelle) ;
- l'instruction *case* (instruction conditionnelle) ;
- l'instruction *for* (boucle).

Affectation

L'affectation est de loin la plus courante. À gauche, nous trouvons un signal, suivi d'un signe égal et à droite une expression.

```
signal = expression;
```

La puissance de cette instruction vient de l'expression. Pour celle-ci, nous disposons d'une quantité d'opérateurs différents pour manipuler les bits. Il s'agit notamment de certains opérateurs mathématiques comme +, –, et *. Il faut noter que pour la division, / ne peut pas être utilisé si l'expression est dynamique. En effet, la division est trop compliquée pour que les outils déterminent une valeur par défaut raisonnable à notre place. La division est toujours possible, elle demande juste un peu plus d'effort et doit être planifiée.

Instruction *if*

L'instruction *if* obéit à la syntaxe habituelle :

```
if (expr) { … } else { … }
```

Si l'expression qui suit le "*if*" est vraie (non nulle), alors la 1ère série de lignes est validée. Si elle est fausse (zéro), les lignes du bloc "else" sont validées. La partie "else" est facultative. Remarque : j'ai écrit "validée" et non "exécutée". On tombe facilement dans ce piège lorsque nous avons des instructions *if* et des boucles *for* à introduire dans la structure du programme. Les *if* sont le plus souvent réalisés par un multiplexeur matériel. Il suffit de sélectionner l'une des deux entrées selon la valeur d'une expression.
Lorsque nous attribuons une valeur à un signal dans un bloc *always* , il faut qu'en toute circonstance une valeur lui soit attribuée. En général, cela signifie que si nous attribuons une valeur à quelque chose dans une instruction *if*, la partie *else* de l'instruction doit avoir son pendant. La seule exception à cette règle est l'entrée *d* d'un type *dff* ou *fsm*. Nous y reviendrons.
Une autre façon de s'assurer que nous attribuons toujours une valeur est de commencer le bloc *always* par quelques valeurs par défaut raisonnables. Examinons le pseudo-code suivant :

```
led = 0;
if (button_pressed)
    led = 1;
```

Lorsque le bouton n'est pas enfoncé, *led* a la valeur 0. Cependant, que se passe-t-il si le bouton est enfoncé ? *led* prend-il la valeur 0 puis est-il mis à jour avec une valeur de 1 ? Eh bien non. Lorsque le bouton est pressé, *led* a toujours la valeur 1. Les affectations en queue d'un bloc *always* ont la priorité sur les affectations de tête. Cela revient à dire que le bloc *always* est *toujours* évalué et cela, *instantanément*.
Maintenant, imaginons que nous n'ayons pas cette valeur par défaut avant le *if*. Quelle valeur aurait *led* bouton non enfoncé ? Il est tentant de penser qu'il conserverait sa valeur précédente, mais souvenons-nous que les signaux ne peuvent pas stocker de valeurs. Ils sont juste comme des fils reliant deux choses entre elles.
Une telle valeur par défaut de 0, pourrait être réalisée avec un circuit de type multiplexeur. Dans ce cas

Premier FPGA : s'amuser avec la MLI

Lorsque vous allumez une carte Alchitry Au [5] ou Alchitry Cu [6] pour la première fois, la configuration FPGA par défaut crée sur les LED un effet de vague amusant. Dans le tutoriel, nous passons en revue les différentes étapes permettant de parvenir à un tel résultat. C'est un très bon guide pour aborder un circuit et se perfectionner avec, compte tenu du fait qu'*in fine* nous travaillons sur du matériel et non du logiciel.

Pour en savoir plus, consultez le tutoriel complet :
https://learn.sparkfun.com/tutorials/first-fpga-project---getting-fancy-with-pwm

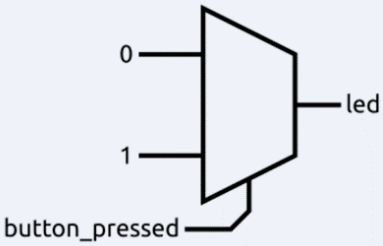

Figure 2 : Bouton de type FPGA pour le contrôle de l'allumage et de l'extinction d'une LED.

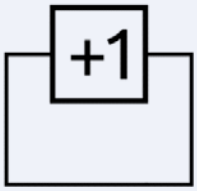

Figure 3 : Fonction d'addition.

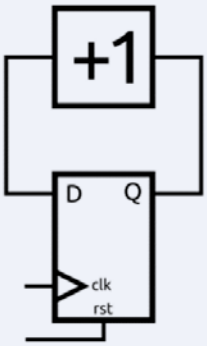

Figure 6 : DFF et compteur combinés.

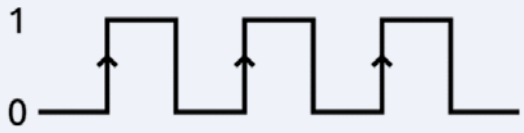

Figure 4 : Signal d'horloge dont les fronts montants sont marqués par des flèches.

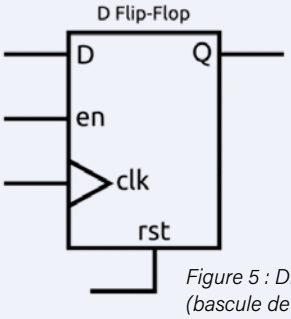

Figure 5 : DFF (bascule de type D).

trivial, cela pourrait être simplifié en se contentant de connecter les signaux *led* et *bouton pressé* ensemble (**fig. 2**).

Instruction *case*

L'instruction *case* obéit à la syntaxe suivante :

```
case(expr) {
    value: statement;
    value: statement;
    default: statement;
}
```

Elle fonctionne exactement comme l'instruction *if*. Elle ne sert qu'à réaliser plusieurs branchements à l'aide d'une seule expression. Chaque partie value de l'instruction doit être une constante. La branche optionnelle défaut est un fourre-tout.

Contrairement à l'instruction case d'un programme informatique, *case* n'offre ici aucun gain de vitesse par rapport à une série d'instructions *if*. Elle améliore la clarté et l'ergonomie du code, c'est tout.

Boucle *for*

Avec Lucid, la boucle *for* adopte la même syntaxe que le C ou le Java, mais il y a quelques restrictions :

```
for (init; eval; increment) { … }
```

La boucle *for* est généralement utilisée avec un type *var* qui sert à stocker des valeurs utilisées dans la description, mais n'apparaissant pas directement dans le circuit. La principale limitation matérielle de la boucle *for* est qu'elle doit avoir un nombre d'itérations défini par une constante. Les outils doivent en effet pouvoir dérouler la boucle. Une boucle *for* consiste à cloner la partie de code correspondante autant de fois que nécessaire. C'est beaucoup plus lisible. En l'absence d'une bonne raison, mieux vaut les éviter. Avec des boucles *for,* on se retrouve facilement avec un grand circuit très lent.

Nombres

Dans Lucid, il y a diverses façons de définir une constante numérique. Le plus simple est de taper le nombre par ex. 14. Lorsque nous voyons un nombre isolé, il est en décimal (en base $_{10}$) et le nombre de bits utilisés pour le représenter est le minimum requis dans un format non signé, à moins qu'il ne soit négatif.

Pour mieux contrôler le nombre de bits utilisés, nous pouvons ajouter un préfixe *xd* où *x* est le nombre de bits à utiliser. Par exemple, *8d14* est la valeur décimale 14 représentée sur 8 bits. En remplaçant le *d* par *h* ou *b* nous exprimerons respectivement le nombre en hexadécimal (base $_{16}$) ou en binaire (base $_{2}$). Avec ces deux formats, nous pouvons spécifier le nombre de bits à utiliser avant le h ou le b. Si nous omettons le nombre de bits, les nombres hexadécimaux utilisent par défaut 4 bits par chiffre. Par exemple, *h08* utilise 8 bits puisqu'il y a deux chiffres alors que la valeur 8 pourrait être représentée avec seulement 4 bits. Pour le binaire, le nombre de bits est simplement le nombre de chiffres lorsqu'il n'est pas explicitement spécifié. Ainsi, *b101001 a une* largeur de 6 bits. Si un nombre décimal est écrit avec un *d,* mais que le nombre de bits est omis, il se comporte comme si le *d* était également omis.

Arrays (tableaux)

Nous rencontrerons de nombreux signaux multibits comme l'entrée *led* de notre module de niveau supérieur. Les bits d'un tableau peuvent être indicés individuellement en utilisant la syntaxe *signal[bit]* où *bit* est une expression. S'il s'agit d'une valeur dynamique, nous devons nous assurer que la valeur sera toujours dans les limites du tableau.

L'accès à des sous-ensembles de bits se fait en utilisant la syntaxe de tableau : *[max:min]*. Ici, la plage de bits de *min* à *max,* (tous deux inclus) est sélectionnée. Avec cette syntaxe, les deux valeurs doivent être des constantes.

Pour sélectionner dynamiquement un sous-ensemble de bits, nous pouvons utiliser la syntaxe *[start+:width]*. Ici, *start* est le bit inférieur à sélectionner et *width* est le nombre total de bits à sélectionner à partir du bit de départ (*start*) inclus. Dans cette syntaxe, seule la *largeur* doit être une constante. Nous pouvons également utiliser la variante : *[start-:width]*. Dans cette syntaxe, *start* est le bit supérieur de la sélection et non le bit inférieur.

Dans Lucid, les tableaux peuvent être multidimensionnels. Il suffit d'ajouter les dimensions voulues dans la déclaration :

```
sig my_array[dim1][dim2][dim3];
```

Les dimensions d'un tableau doivent être déclarées avec des valeurs constantes. Nous pouvons ensuite indicer le tableau avec des sélecteurs comme ci-dessus. Notons que nous pouvons utiliser les sélections de sous-tableaux seulement comme dernier sélecteur.

Logique séquentielle et DFF

Ici cela devient vraiment intéressant. La logique combinatoire est très importante, mais un système sans état ni mémoire est assez limité. Alors comment créer une mémoire ? En gros, nous avons juste besoin d'une sorte de boucle de rétroaction. Pour créer un compteur, il suffit d'ajouter 1 au résultat de la dernière addition. Le problème est de savoir comment contrôler cette boucle. À première vue, cela peut sembler trivial, mais y regarder de plus près revient à ouvrir la boîte de Pandore. En effet, nous verrons que rien ne fonctionne comme prévu. Nous pourrions créer ce compteur avec un additionneur dont l'une des valeurs d'entrée serait fixée à 1. Pour simplifier, logeons-le dans un seul bloc (**fig. 3**). Si nous connectons son entrée à la sortie (notre boucle), nous pensons créer un compteur incrémentiel.

Mais des questions se posent d'emblée. À quelle valeur ce compteur commence-t-il et à quelle vitesse compte-t-il ? La valeur initiale dépend de la façon dont l'alimentation du circuit a été appliquée et de la disposition du circuit. Elle dépendrait sans doute de la température et d'autres facteurs environnementaux. Il serait désastreux que notre circuit se comporte différemment selon qu'il pleuve ou qu'il vente.

Le pire, c'est que ce circuit ne fonctionnerait même pas. En effet, un circuit d'addition, comme la plupart des logiques combinatoires à sorties multibits, produit des résultats intermédiaires erronés. Dans le cas d'un additionneur, le bit le moins significatif est calculé en premier et chaque bit suivant utilise le résultat du bit qui le précède. Comme on n'attend pas que le résultat soit valide, les valeurs intermédiaires erronées sont renvoyées dans l'additionneur qui propage alors d'autres valeurs erronées et notre bel objet ne produit que des résultats faux. Comment régler ce problème ? Il nous suffirait de pouvoir contrôler la chronologie de la boucle de rétroaction. C'est justement ce que font les bascules de type D ou DFF (D *flip-flop*).

Avant de voir le DFF en détail, parlons des horloges. Une horloge est un signal qui passe de 0 à 1 puis de 1 à 0, indéfiniment, à une fréquence donnée (**fig. 4**). L'horloge des cartes Alchitry bascule 100 millions de fois par seconde, soit 100 MHz. Grâce à ce signal périodique, nos circuits vont percevoir le temps. La transition de 0 à 1, à savoir le front montant, constitue généralement la partie active du signal. Dans l'illustration, ces fronts sont matérialisés par des flèches.

Revenons au DFF (**fig. 5**). Les DFF sont un type de mémoire. Ils ont une entrée, D, et une sortie, Q. Lorsque leur entrée d'horloge passe de 0 à 1, la valeur de D est mémorisée et renvoyée sur Q jusqu'au prochain front montant de l'horloge. Peu importe que D change *entre* deux fronts montants consécutifs de l'horloge, Q ne change pas. Dans la figure 5, le DFF est dessiné avec les signaux optionnels d'activation (en = *enable*) et de réinitialisation (rst = *reset*). Le signal *en* peut être utilisé pour empêcher le DFF de copier une nouvelle valeur sur un front montant. Le signal *rst* est utilisé pour forcer la valeur de Q à une valeur connue. Les DFF des FPGA sont prévus pour être réinitialisés à 0 ou 1. Nous utilisons le DFF de notre compteur pour contrôler la boucle (**fig. 6**). La valeur initiale de Q, p.ex. 0, sera fixée par *rst*. Cela signifie que nous savons que Q vaut 0. Alors D passe à 1. Sur le front montant suivant de l'horloge, Q prend la valeur de D. Par conséquent Q passe à 1 et D passe à 2. Sur chaque front montant, Q augmente de 1. C'est ce que nous voulions ! La fréquence de l'horloge détermine la vitesse d'incrémentation de notre compteur, non sans quelques limitations. L'horloge doit être suffisamment lente pour que la valeur de D ait le temps de s'actualiser après un changement de Q. Nous excluons que notre DFF sauvegarde une des valeurs intermédiaires invalides que l'additionneur produit. Le temps nécessaire pour que la valeur de sortie de l'additionneur soit valide s'appelle le délai de *propagation.* C'est le temps qui s'écoule entre un changement des entrées et la stabilisation de la sortie. Plus nous ajoutons de logique, plus ce délai est long. Le délai dépend aussi de la technologie de fabrication du circuit. Les outils disposent de modèles pour chaque FPGA et si nous indiquons la fréquence d'horloge que nous utilisons, ils tenteront de réaliser notre projet de sorte que les exigences en matière de délai soient respectées. Dans cet exemple, nous avons un ensemble de DFF inséré pour boucler un bloc de logique combinatoire. Il est plus courant que la sortie d'un ensemble de DFF soit envoyée vers un autre ensemble de DFF via un bloc de logique combinatoire, créant ainsi un pipeline. Dans tous les cas, c'est le plus long délai de propagation du circuit qui fixe la fréquence maximale de l'horloge. En décomposant notre circuit en blocs de logique combinatoire ayant un délai de propagation voisin, nous pouvons optimiser la fréquence d'horloge. La synchronisation peut être laborieuse, mais en général, nous pouvons nous en sortir en utilisant la même horloge pour tout le circuit. Pour peu que nous n'ayons pas de circuit trop long à parcourir, les outils résoudront le problème pour nous. À 100 MHz, nous pouvons en pratique faire beaucoup de choses entre les DFF. Les problèmes ne surviennent que si nous essayons d'enchaîner trop de choses ou d'inclure trop de multiplications.

Le respect de la chronologie devient plus ardu à l'approche de la limite des ressources du FPGA. Comme les outils doivent intégrer de plus en plus de choses dans un FPGA de taille donnée, les possibilités de configuration se réduisent, ce qui peut les empêcher d'atteindre la vitesse nécessaire.

Listage 3.

```
module blinker (
   input clk,  // clock
   input rst,  // reset
   output out
 ) {

  always {
    out = 0;
  }
}
```

Travaux pratiques : faire clignoter une LED

Appliquons maintenant ce qui précède dans un projet de démonstration qui fera clignoter une LED. Créons un autre projet dans Alchitry Labs [4] et choisissons *Base Project* sur le menu déroulant *From Example*, (**fig. 7**). Cliquons sur l'icône *nouveau fichier* de la barre d'outils (icône

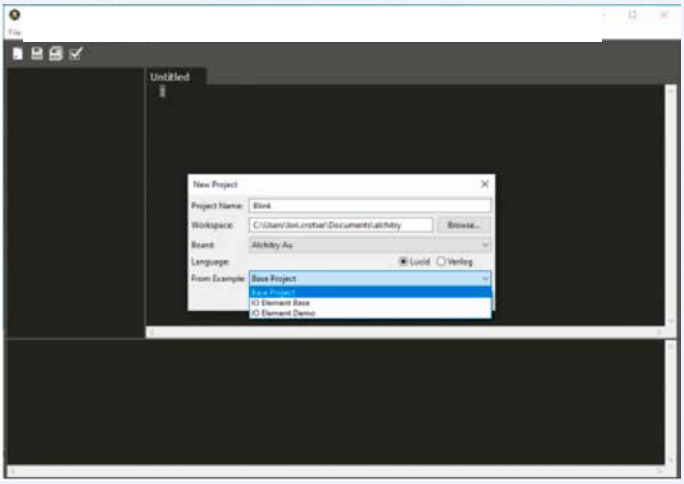

Figure 7 : Sélection du projet de base dans le logiciel Alchitry Labs.

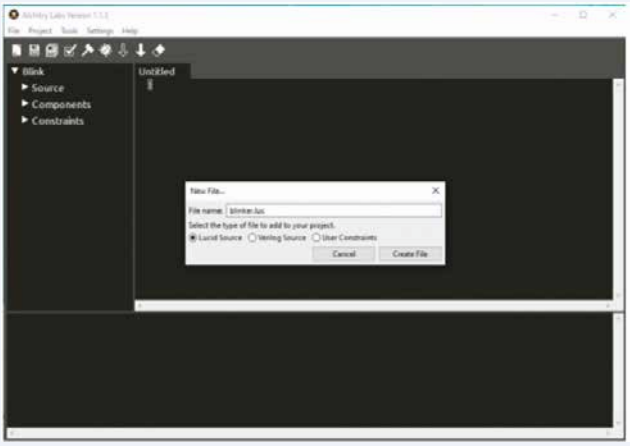

Figure 8 : Création d'un nouveau fichier source Lucid.

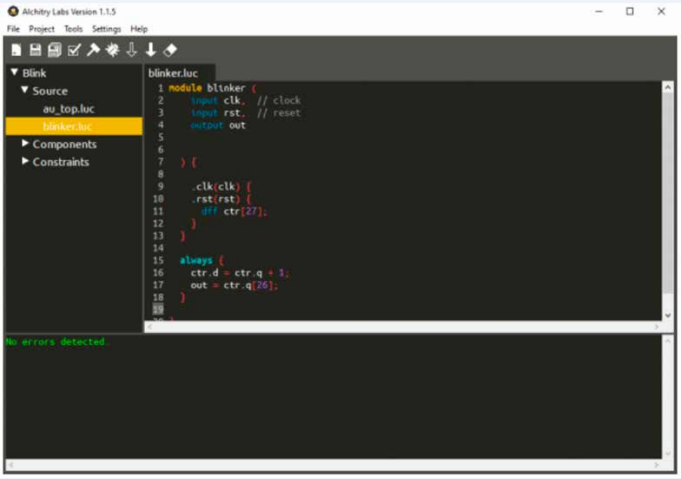

Figure 9 : Le "module" créé à partir du programme dans le listage 3.

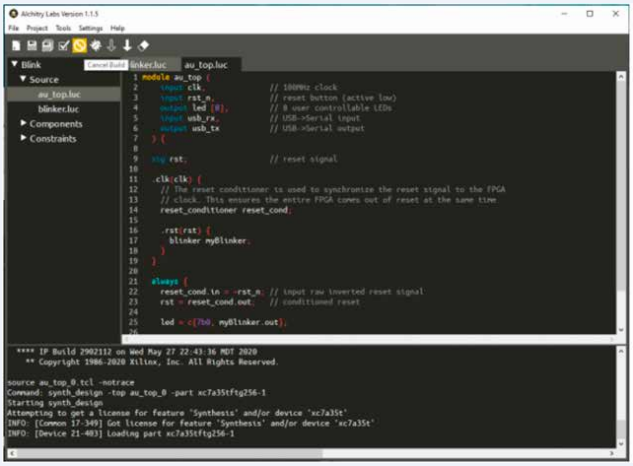

Figure 10 : Téléchargement du projet sur le tableau d'Alchitry.

la plus à gauche) et créons un nouveau fichier *Lucid Source* nommé *blinker.luc* (**fig. 8**). Cela va créer un module de base qui ressemble au **listage 3** pour le code et à la **figure 9** pour l'écran.

Le modèle de module par défaut ajoute les entrées d'horloge et de réinitialisation et une sortie factice pour le moment. Pour faire clignoter une LED, il faut créer un compteur qui sera utilisé pour basculer la LED. Si la LED bascule à chaque cycle d'horloge, le clignotement est trop rapide pour être visible.

Pour faciliter la tâche, nous définissons les blocs de connexion pour l'horloge (*clk*) et la réinitialisation (*rst*) puis déclarons le DFF à l'intérieur de ceux-ci :

```
.clk(clk) {
  .rst(rst) {
    dff ctr[27];
  }
}
```

Cela crée un ensemble de 27 DFF nommés *ctr*. Nous insérons ensuite le DFF dans le bloc "*always*".

```
always {
  ctr.d = ctr.q + 1;
  out = ctr.q[26];
}
```

Pour accéder aux signaux d'un module ou d'un DFF, nous utilisons la notation pointée. La première ligne du bloc *always* relie l'entrée D des DFF à la sortie Q plus 1. La valeur de *ctr.q* s'incrémentera ainsi une fois par cycle d'horloge. Notons que le signal *d* est en écriture seule et que *q* est en lecture seule. La 2e ligne prend le bit le plus significatif, le n° 26, et le connecte à la sortie. Comme *ctr* a une largeur de *27* bits, il a des indices de 0 à 26. Comme *ctr* s'incrémente une fois par cycle d'horloge, un nombre de 27 bits peut contenir 227 = 134 217 728 valeurs différentes. De plus, notre fréquence d'horloge étant de 100 MHz, *ctr* déborde une fois toutes les 1,34 s. Durant la première moitié de ce cycle, le bit le plus significatif sera 0 puis 1 durant la seconde moitié. En connectant ce bit à la sortie, nous basculerons la sortie toutes les 0,67 s.

Passons maintenant au module *top level* et instancions notre nouveau module. Notons que j'utilise une carte Au. Le module aurait la même apparence pour la Cu, mais le nom serait *cu_top* et non *au_top* (**listage 4**). Ici, j'ai ajouté un bloc de connexion pour le signal de réinitialisation et créé une instance du module clignotant appelée *myBlinker*. Ensuite, dans le bloc *always*, je l'ai connecté à la sortie *led* en utilisant la syntaxe de *concaténation.* Les éléments à l'intérieur de *c{...}* sont collés ensemble pour former un seul tableau. Dans notre cas, nous concaténons donc sept 0 avec le bit de *myBlinker.out*. Cela va éteindre les 7 LED de poids fort et connecter notre signal de clignotant à la LED de poids faible. Nous pouvons maintenant réaliser le projet en cliquant sur l'icône du marteau, puis le charger dans notre carte en

Listage 4.

```
module au_top (
    input clk,          // 100MHz clock
    input rst_n,        // reset button (active
low)
    output led [8],     // 8 user controllable
LEDs
    input usb_rx,       // USB->Serial input
    output usb_tx       // USB->Serial output
  ) {

  sig rst;              // reset signal

  .clk(clk) {
    // The reset conditioner is used to
    // synchronize the reset signal to the FPGA
    // clock. This ensures the entire FPGA comes
    // out of reset at the same time.
    reset_conditioner reset_cond;

    .rst(rst) {
      blinker myBlinker;
    }
  }

  always {
    reset_cond.in = ~rst_n; // input raw inverted
reset signal
    rst = reset_cond.out;   // conditioned reset

    led = c;

    usb_tx = usb_rx;        // echo the serial data
  }
}
```

Listing 5.

```
module blinker #(
    MAX_VALUE = 50000000 : MAX_VALUE > 0
  )(
    input clk,  // clock
    input rst,  // reset
    output out
  ) {

  .clk(clk) {
    .rst(rst) {
      dff ctr[$clog2(MAX_VALUE)];
      dff led;
    }
  }

  always {
    ctr.d = ctr.q + 1;
    if (ctr.q == MAX_VALUE-1) {
      ctr.d = 0;
      led.d = ~led.q;
    }

    out = led.q;
  }
}
```

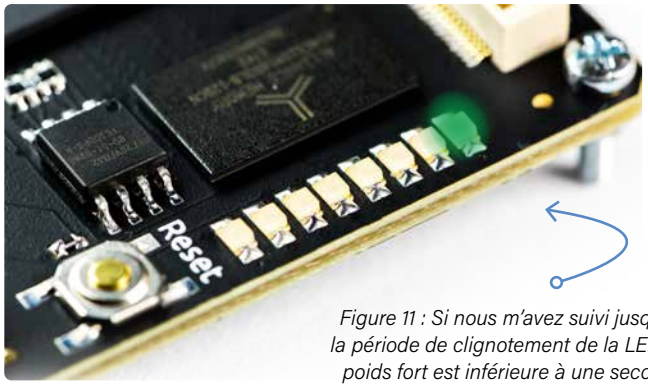

Figure 11 : Si nous m'avez suivi jusqu'ici, la période de clignotement de la LED de poids fort est inférieure à une seconde.

appuyant sur l'icône de la flèche pleine vers le bas (**fig. 10**). La LED du haut devrait maintenant clignoter un peu plus lentement qu'une fois par seconde (**fig. 11**).

Amélioration des modules

Nous pouvons améliorer notre clignotant. Tout d'abord, la LED clignote avec une période inhabituelle. Ramenons-la à 1 s exactement en comptant jusqu'à 50 000 000 avant d'allumer la LED. Pour ce faire, nous avons besoin d'un autre DFF pour mémoriser l'état de la LED.

```
  .clk(clk) {
    .rst(rst) {
      dff ctr[28];
      dff led;
    }
  }
```

Dans le bloc "*always*", nous vérifions que *ctr.q* vaut 49 999 999 (de 0 à 49 999 999, il y a 50 000 000 incréments) et si c'est le cas, nous le remettons à 0 et basculons la LED.

```
  always {
    ctr.d = ctr.q + 1;
    if (ctr.q == 49999999) {
      ctr.d = 0;
      led.d = ~led.q;
    }

    out = led.q;
  }
```

Ici, j'ai utilisé l'opérateur d'inversion de bits, le tilde : ~. Il permet d'inverser chaque bit d'un signal. Comme *led.q* n'a qu'un seul bit, l'opération n'inverse que ce bit.

Si nous avons bonne mémoire, nous savons qu'en toutes circonstances un signal doit avoir une valeur, sauf pour les DFF. Comme les DFF peuvent mémoriser leur valeur, si l'entrée *d* ne reçoit pas d'affectation pendant un cycle d'horloge, la valeur du DFF ne changera pas.

Notre module ne stocke plus que les valeurs de 0 à 49 999 999 dans *ctr*, mais il s'agit toujours d'un tableau de 28 bits. C'est du gaspillage, car il ne faut que 26 bits pour stocker nos valeurs. Nous pourrions juste réduire à 26 la taille du tableau, mais si nous voulons changer la valeur maximale, il faudrait recalculer cette valeur. Des fonctions Lucid peuvent se charger de ce calcul pour nous. Par ex. *$clog2(50000000)* qui calculera la valeur plafond du log base 2 de la valeur donnée. Cela équivaut à "combien de bits faut-il pour stocker un tel nombre". Dans notre cas, il trouvera 26.

```
dff ctr[$clog2(50000000)];
```

En ce qui concerne le changement de la valeur maximale, nous modifions notre module pour qu'il l'accepte comme paramètre afin

Sur la vague FPGA

L'internet offre de nombreuses ressources sur les FPGA. Voici quelques liens choisis pour stimuler votre intérêt.

- Alchitry, "Using FPGAs", www.sparkfun.com/fpga.
- J. Rajewski, "How Does an FPGA Work?", SparkFun, https://learn.sparkfun.com/tutorials/how-does-an-fpga-work
- J. Rajewski, "First FPGA Project: Getting Fancy with PWM", SparkFun, www.elektormagazine.fr/esfe-en-fpga2
- J. Rajewski, "External IO and Metastability", SparkFun, https://learn.sparkfun.com/tutorials/external-io-and-metastability
- S. Cass, "Painless FPGA Programming", IEEE Spectrum, November 2020, https://spectrum.ieee.org/geek-life/hands-on/painless-fpga-programming

de pouvoir la spécifier lors de l'instanciation du module. Il suffit d'ajouter une *liste de paramètres à* notre module. Cette liste précède la liste des *ports*.

```
module blinker #(
    MAX_VALUE = 50000000 : MAX_VALUE
> 0
  )(
    input clk,  // clock
    input rst,  // reset
    output out
  ) {
```

La syntaxe spécifiée pour la liste des ports est : #(param, param, param). Chaque paramètre peut n'être qu'un simple nom (en capitales) ou plus complexe, comme dans notre exemple où nous avons fixé une valeur par défaut de 50 000 000. L'usage de valeurs par défaut est conseillé, sauf quand on veut forcer la spécification d'une valeur à l'instanciation. Après l'affectation par défaut, nous pouvons spécifier une condition à laquelle le paramètre doit répondre. Elle doit utiliser le paramètre lui-même et peut aussi utiliser tous les paramètres déclarés avant lui. L'évaluation de cette condition doit donner "*true*" (vrai). Si ce n'est pas le cas, une erreur est signalée lors de l'instanciation. Cela permet d'écrire le module avec quelques hypothèses sur la valeur du paramètre et de savoir qu'elles seront respectées. Nous pouvons alors remplacer les occurrences de 50 000 000 dans notre module par `MAX_VALUE` (**listage 5**).

Si nous compilons et chargeons le projet maintenant, la LED doit clignoter à un rythme d'une fois par seconde. Cependant, si nous revenons au module de niveau supérieur, nous pouvons modifier l'instanciation comme ceci :

```
blinker myBlinker(#MAX_VALUE(25000000));
```

Maintenant, si nous compilons et chargeons le projet, la LED clignotera deux fois par seconde.

En résumé

Vous voilà au bout de vos peines. Les circuits FPGA sont constitués de blocs de logique combinatoire qui effectuent tous les traitements, et de DFF qui mémorisent les valeurs et contrôlent le flux de données. Les circuits eux-mêmes sont décomposés en modules. Les modules peuvent être utilisés par d'autres modules et peuvent avoir des paramètres pour personnaliser chaque instanciation de ceux-ci. Chaque fois qu'un module est instancié, le circuit qui lui est destiné est dupliqué dans le FPGA.

Lors de la conception de matériel, il est important de réfléchir à la manière dont ce projet pourrait être mis en œuvre pour créer des circuits efficaces. Dans le cas de notre clignotant, sans nous préoccuper de la vitesse de clignotement, la première version du module nécessiterait beaucoup moins de ressources pour sa mise en œuvre. En effet, il n'a pas besoin d'un comparateur pour vérifier la valeur du compteur. Il continue simplement à ajouter 1 et utilise le débordement naturel de l'addition binaire pour initialiser le compteur.

200646-03

Alchitry

Alchitry et Justin Rajewski

Soutenu sur KickStarter, Alchitry est l'une des sources les plus complètes et les mieux informées sur les FPGA. Séduite par les cartes Alchitry Au et Alchitry Cu, l'équipe de SparkFun a lancé une collaboration avec Justin Rajewski, fondateur et ingénieur à tout faire d'Alchitry. Aujourd'hui, Justin se concentre sur le développement de nouveaux produits, la construction de projets et la génération de contenu. Il a tiré parti de l'expérience de SparkFun en matière de production et de logistique pour fabriquer ses cartes. Justin apporte à SparkFun son expertise inégalée en matière de FPGA.

WEBLINKS

[1] SparkFun, "Using FPGAs" : https://www.sparkfun.com/fpga
[2] Wikipedia, "Hardware Description Language" : https://en.wikipedia.org/wiki/Hardware_description_language
[3] Lucid: https://alchitry.com/pages/lucid-fpga-tutorials
[4] Alchitry Labs: https://alchitry.com/blogs/tutorials/getting-started-with-the-au
[5] carte Alchitry Au: https://www.elektormagazine.fr/esfe-en-fpga1
[6] carte Alchitry Cu: https://www.sparkfun.com/products/16526