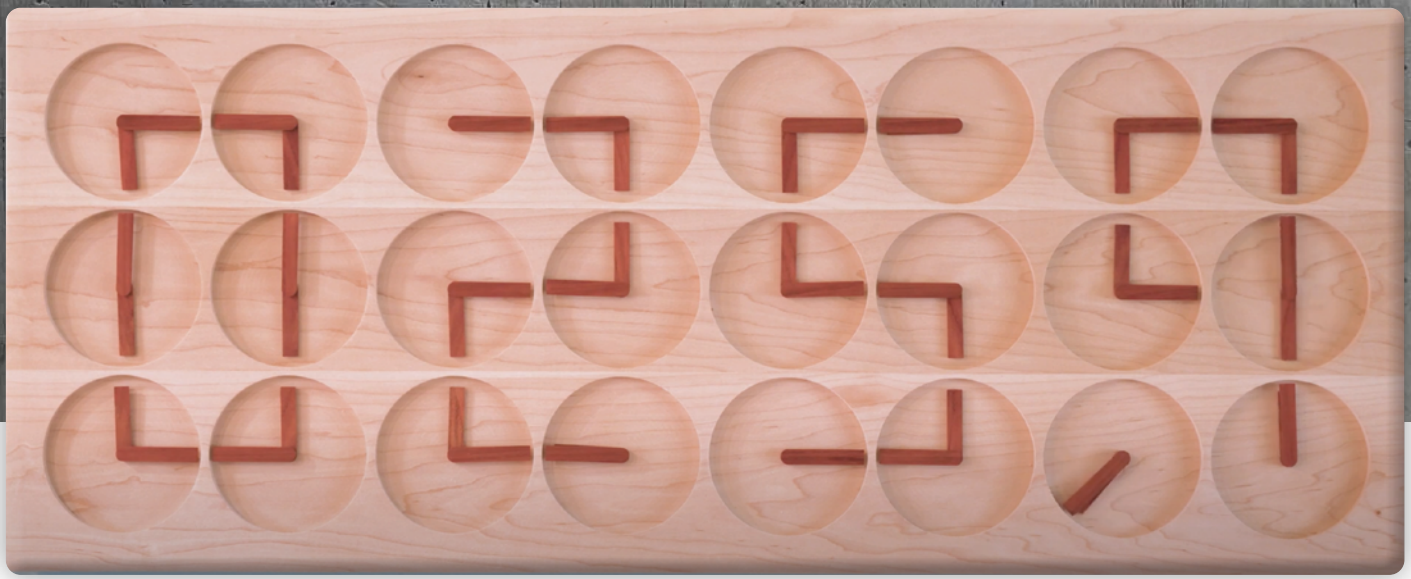
CLOCKCLOCK

horloge récursive

Arduino donne l'heure avec un FPGA



Justin Rajewski (Alchitry) (États-Unis)

Quelle heure est-il ?
L'heure d'attaquer un magnifique projet Alchitry !
Cet article va vous guider dans la construction d'une Clockclock dont les moteurs sont commandés par la carte de développement FPGA d'Alchitry Au.

Qu'est-ce qu'une *ClockClock* ? C'est 'une horloge faite d'horloges ! Une méta-horloge en somme. Pour former chacun des (grands) chiffres de l'heure affichée par *ClockClock*, j'utilise plusieurs horloges à aiguilles ordinaires. L'idée n'est pas de moi. Je suis tombé sur ce concept astucieux il y a déjà des années et j'ai toujours pensé que ce serait un excellent projet de démonstration de FPGA tant il nécessite de signaux de contrôle. Vous trouverez l'horloge originale de *Humans since 1982* ici [1].

Ce projet est une excellente démonstration du FPGA pour différentes raisons. D'abord, la *ClockClock* nécessite 48 moteurs pas à pas pour mouvoir les 2 aiguilles indépendantes de ses 24 horloges. L'utilisation d'une commande standard implique deux signaux (pas + sens) de commande par moteur soit 96 sorties. Pour économiser l'énergie, mieux valait pouvoir désactiver les pilotes quand l'horloge est immobile. Cela a ajouté quatre sorties supplémentaires (une pour chaque «chiffre» de l'horloge). Je voulais également utiliser un Arduino pour créer les animations, plus faciles à obtenir en programmation qu'en matériel. J'ai choisi I²C pour dialoguer avec l'Arduino via le connecteur *Qwiic* de l'Alchitry Au, donc 2 broches de plus. Au total nous atteignons 102 E/S. Or, la carte Alchitry Au a exactement 102 broches d'E/S [2].

Ce projet a le mérite non seulement d'épuiser la foule d'E/S du FPGA, mais aussi d'utiliser le connecteur *Qwiic* du FPGA d'une manière peu conventionnelle : ici, le FPGA agit comme périphérique et non comme contrôleur. L'Arduino le contrôle et lui envoie toutes les commandes. Ce sera un modèle utile pour de nombreux projets. Certaines tâches, très simples sur le plan du logiciel, peuvent nécessiter un matériel très complexe. L'inverse est également vrai. En associant un microcontrôleur et un FPGA, nous obtenons le meilleur des deux mondes. Le connecteur *Qwiic* des deux cartes facilite la tâche.

Matériel et outils nécessaires

Pour suivre cet article, vous aurez besoin du matériel énuméré dans l'encadré **Accessoires**. Vous n'aurez peut-être pas besoin de tout, selon ce que vous avez déjà. Composez votre panier, lisez le guide et rectifiez le panier si nécessaire.
Il y a plusieurs façons d'usiner les pièces pour ce projet. Voici les outils que j'ai utilisés :

- imprimante 3D ;
- fraiseuse CNC Shapeoko XXL ;
- scie à ruban, raboteuse et ponceuse orbitale pour le bois.

Et aussi :

- 48× moto-réducteurs pas-à-pas [3] ;
- 48× modules drivers de moteur pas à pas StepStick avec dissipateur thermique [4] ;
- 1× UBEC BEC ajustable UBEC 2-6S pour drone quadcopter RC [5] ;
- 1× Alimentation à découpage AC-DC sous boîtier [6].

Suggestions de lecture

Si vous connaissez mal le système *Qwiic*, nous vous conseillons de lire la documentation sur le site www.sparkfun.com/qwiic. Nous vous conseillons également de consulter ces tutoriels avant de continuer :

- *Programmation d'un FPGA* [7]. Notions élémentaires pour s'approprier les FPGA.
- *Comment un FPGA fonctionne-t-il ?* [8]. Le b.a.-ba du FPGA (Field Programmable Gate Array = réseau de portes programmable in-situ).
- *Premier projet FPGA - Getting Fancy with PWM* [9]. 1er projet utilisant le FPGA embarqué d'Alchitry pour manipuler la modulation PWM.

Construction physique

Je serai bref car mon sujet est le FPGA, pas le travail du bois. Suivez les liens ci-dessous pour accéder aux fichiers du projet et les télécharger.

- Fichier CAO (Fusion 360) [10].
- Alchitry (FPGA) [11].
- Code Arduino (ZIP) [12]

Pour débuter ce projet il fallait imaginer comment fabriquer l'un des mouvements de l'horloge. J'avais besoin de deux moteurs pas à pas et d'un moyen de les coupler à deux arbres de sortie concentriques. J'ai d'abord trouvé sur Amazon ces moteurs pas à pas miniatures (à peine 8 × 9,2 mm !). J'ai conçu un réducteur double (2 couronnes et 2 pignons) emboîté sur les moteurs (**fig. 1**).

Malheureusement, ces petits moteurs ne parvenaient pas à faire tourner les engrenages. Leur couple est minuscule et, quand j'ai appliqué une tension d'alimentation suffisante pour qu'ils tournent, ils ont chauffé au point de faire fondre les pièces en PLA imprimées en 3D.
J'y ai renoncé, et j'ai commandé un lot de moteurs pas à pas 28BYJ-48 [13], un peu plus gros mais assez petits pour l'horloge. Leur réducteur interne offre un couple fort. Qui dit réducteur interne dit aussi couple résistant interne suffisant pour conserver la position atteinte après coupure du moteur.
Le mouvement que j'ai conçu est à entraînement direct de l'aiguille des minutes et l'aiguille des heures est entraînée par un engrenage, ainsi le moteur est déporté latéralement.
Je raconte la suite de la construction matérielle de notre *ClockClock* en images assorties de brefs commentaires. Profitez du spectacle !

FPGA

Avant de commencer un projet FPGA, quel qu'il soit, il est important de définir ce que nous en attendons. Dans le cas présent, je devais créer un circuit acceptant des commandes par I²C et faire tourner les moteurs en conséquence.
J'ai opté pour les 2 commandes suivantes : un nombre de pas et une valeur équivalant à la période entre pas. Au départ, j'avais pensé rendre le contrôleur plus souple en modulant automatiquement le nombre de pas, mais cela s'est révélé inutile et ne faisait que compliquer la coordination entre les aiguilles.
Je voulais aussi constituer une file d'attente pour une série de commandes. La synchronisation du *Qwiic* n'aurait pas d'importance, car chaque commande serait exécutée l'une après l'autre.
Il fallait enfin que le projet résolve la chronologie de l'envoi des pas pour activer/désactiver les moteurs en conséquence et économiser l'énergie.

L'animateur

Pour commencer, j'ai créé un module contrôlant un seul moteur pas à pas. Ce module accepte une commande permettant de faire un nombre de pas donné avec une temporisation donnée entre chaque pas. Il délivre ensuite les signaux de sens et de pas appropriés pour la commande du moteur pas à pas. Le code du module est donné par le **listage 1**.
La première chose à noter est que le contrôleur pas à pas utilisé exige que l'entrée du sens soit stable pendant au moins 200 ns avant et après le front montant de l'entrée du pas. Il exige également que

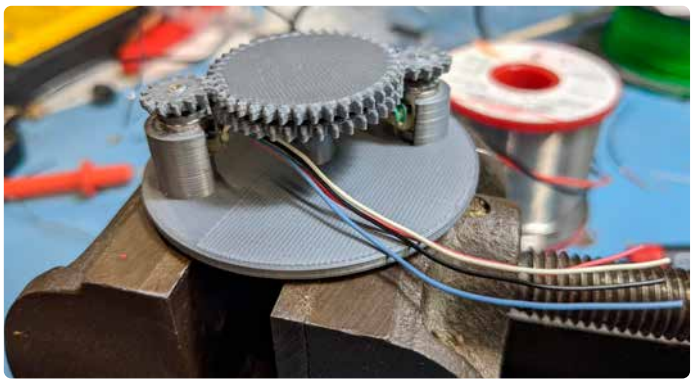


Figure 1 : Moteur pas à pas miniature (Amazon) avec 2 couronnes et 2 pignons montés à la presse.

l'impulsion de pas ait une durée minimale haute ou basse de 1 µs. La largeur d'impulsion des pas est facilement obtenue avec le module `pulse_extender` de la bibliothèque de composants. Ce module reçoit des impulsions isolées et les prolonge jusqu'à la durée spécifiée. J'ai fixé la durée à 2 µs par commodité et sécurité. Dès qu'une nouvelle commande d'animation est reçue, la sortie sens est définie et le module attend que le dépassement du `dirCt`. Ce compteur contient 256 valeurs et avec une horloge à 100 MHz, cela signifie qu'il attend 2,56 µs. C'est nettement plus que les 200 ns nécessaires, mais cela garantit la synchronisation avec les longs fils. Le resserrement de la synchronisation ici n'augmenterait pas non plus les performances.

Le changement d'état du pas incrémente un compteur et fait un pas à chaque dépassement. Ce compteur a un prédiviseur par 8, de sorte que chaque incrément de `delayCycles` dans la commande d'animation ajoute 256 cycles de temporisation. Ce prédiviseur permet de limiter `delayCycles` à 16 bits tout en préservant une gamme de vitesses étendue. Même avec ce prédiviseur, j'ai trouvé que le `delayCycles` le plus bas utilisable sans risque est d'environ 760. Cela correspond à 8 s pour une rotation complète.

Porte d'activation (*enable gate*)

Le module suivant est la porte d'activation. Ce module commande activation et blocage du drapeau *nouvelle animation* des animateurs pendant l'activation des moteurs. Il s'assure aussi qu'après l'animation, les moteurs restent activés assez longtemps pour que le dernier pas soit achevé.

Le module prend en charge les drapeaux de *nouvelle animation* en attente pour 12 des moteurs. Il active ensuite les moteurs et attend 42 ms, afin que l'électronique remette les moteurs sous tension et que les moteurs se stabilisent. Après cette période, il permet au drapeau *animation en attente* de passer aux animateurs.

Tant qu'un animateur du groupe fonctionne, il maintient les moteurs activés. Une fois que le dernier a terminé, il maintient les moteurs activés pendant encore 42 ms avant de les désactiver. Pour le code du module, voir le **listage 2**.

Si le module est inactif, `onCtr` est à 0 et `offCtr` est au maximum. Si une *animation en attente* est détectée (la FIFO n'est pas vide), la sortie *enable* est activée et `onCtr` est incrémenté à chaque cycle. Dès que `onCtr` est plein, les drapeaux `new_animation` sont transmis. Dès que toutes les animations sont réalisées, `offCtr` est incrémenté. Dès sa valeur maximale atteinte, `onCtr` est réinitialisé, ce qui désactive les moteurs.

Une ligne intéressante à regarder est la première du bloc *always* :

```
running = |(animator_busy | ~fifo_empty);
```

Sans pratique des opérateurs de réduction bit à bit, cette ligne paraîtra absconse. Son but est de prendre les 12 signaux `animator_busy` et les 12 signaux `fifo_empty` et de les transformer en un seul bit. Raisonnons d'abord sur un cas simple. Un seul moteur fonctionne si la FIFO n'est pas vide ou si elle est occupée à cet instant. L'expression `animator_busy | ~fifo_empty` résoud ce cas. La barre verticale | ou *pipe* (= tuyau) en anglais) est l'opérateur OU bit à bit. Il réalise simultanément et indépendamment un OU entre les bits de même rang des deux opérandes. Le tilde (~) réalise l'inversion de chaque bit. Il inverse ici chacun des bits de `fifo_empty`.

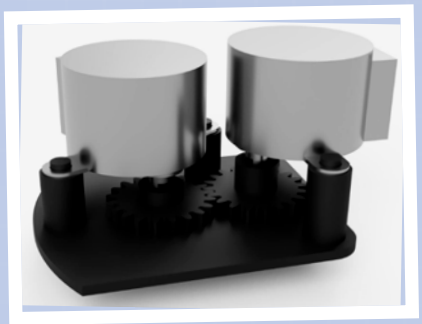
Le résultat de ces opérations est un signal de 12 bits de large qui indique quand chaque animateur est en train de tourner. Cependant, nous devons le condenser en un seul bit. À cet effet, l'opérateur de réduction OR est utilisé. L'opérateur pipe, s'il est placé **seul** devant une valeur, exécute un OU entre **tous** les bits du signal pour les **réduire** à un seul bit. Dans notre cas, cela signifie que si un ou plusieurs des moteurs tournent, running vaut 1.

Plus loin dans le module, j'utilise l'opérateur de réduction AND pour vérifier si tous les bits d'un signal sont à 1 (la *valeur maximale*). Cela fonctionne de la même manière que l'opérateur de réduction OR, mais exécute un ET entre tous les bits. Le résultat est 1 s'ils valent

L'horlogerie de ClockClock

Cette ébauche d'horloge élémentaire montre les deux arbres de sortie. L'arbre central plus long est directement couplé au moteur aligné avec lui. L'arbre extérieur porte une couronne entraînée par un pignon monté sur l'arbre du 2e moteur. Les aiguilles des heures et minutes sont montées par friction sur ces deux arbres.

La friction douce permet de repositionner les aiguilles de l'horloge dans une position neutre avant de la mettre sous tension. C'est important car même si un moteur pas à pas permet une commande de déplacement précise, on ignore toujours la position d'origine.



Les moteurs, simplement collés (cyanoacrylate) sur des entretoises. Toutes les pièces ont été imprimées en PLA noir sur ma Prusa MK3S.

Le gros avantage de l'Au, c'est le bus Qwiic qui permet de communiquer avec un microcontrôleur

tous 1 et 0 dans tout autre cas. L'accent circonflexe (^) est utilisé pour effectuer une réduction XOR (OU exclusif) qui vaudra 1 s'il y a un nombre impair de 1.

Qwiic

Nous allons maintenant nous intéresser au module au_top qui s'occupe de l'interface *Qwiic* et qui coordonne tout. Décortiquons-le **listage 3**.

L'interface *Qwiic* est gérée par le module `i2c_peripheral`. Ce module est un peu compliqué puisqu'il décompose les signaux de démarrage/arrêt et demande qu'on lui indique quand il doit accepter ou envoyer des données.

Dans notre cas, nous pouvons le simplifier en nous contentant de lire les données. Les drapeaux importants deviennent `rx_valid` qui nous indique qu'un nouvel octet a été lu et *stop* qui indique que la transaction I²C a été arrêtée et que nous devons la réinitialiser. La sortie `rx_data` a la valeur de l'octet lu lorsque rx_valid vaut 1.

Si vous voulez réagir à quelque chose, vous devez surveiller les drapeaux start, next, et write. Au cycle d'horloge suivant, vous pouvez mettre `tx_enable` à 1 et fournir des données à envoyer sur `tx_data`. Le module écrira alors cet octet au lieu d'en scruter un.

Le drapeau de départ signale que votre identifiant ID a été détecté sur le bus. En même temps qu'il est activé, write vous dira si le dernier bit de l'octet ID indiquait une lecture (0) ou une écriture (1). Là encore, nous pouvons ignorer tout cela pour ce projet.

Dans le protocole utilisé pour chaque transaction, le 1er octet est l'adresse suivie des données de la commande. Pour les adresses 0 à 47, 4 octets sont attendus. Les 2 premiers codent le nombre de pas et les 2 autres, la valeur de la tempo. L'adresse 8hFF est particulière car elle n'attend qu'un octet après elle et est utilisée pour allumer les LED sur l'*Au*. C'est utile pour tester le bus *Qwiic*.

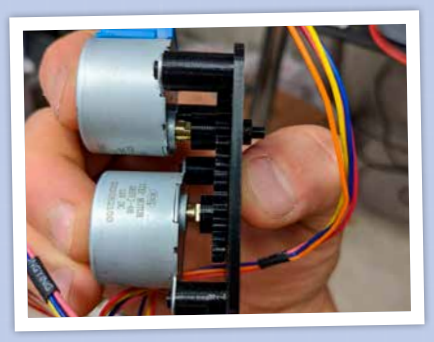
J'ai également fait en sorte que vous n'ayez pas besoin de démarrer/arrêter la transaction I²C pour chaque animation. Chaque paquet de 5 octets est une animation valide, qui tournera en boucle après la réception du dernier octet. Cela permet d'envoyer aux 48 moteurs une nouvelle animation en une seule transaction.

FIFO

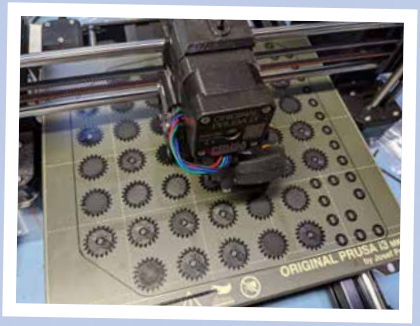
Ce projet comporte 48 FIFO pour conserver des animations supplémentaires pendant que les animateurs sont occupés. Celles-ci sont créées à partir du composant `fifo` de la bibliothèque de composants. Chaque FIFO a une largeur de 32 bits et une profondeur de 128 entrées. Les 32 bits sont répartis en 2 x 16 entre le délai et le nombre de pas. Note : 128 entrées c'est trop pour l'utilisation actuelle, mais cela permettrait d'empiler de nombreuses animations courtes plus rapides et/ou avec gradients de vitesse. L'Au a de toute façon largement assez de RAM intégrée pour le faire.

La FIFO est de type FWFT (*First-Word Fall-Through*) : si *empty* vaut 0, une donnée suit et la valeur de cette donnée est déjà disponible sur `dout`. Mettre `rget` à 1 supprime l'entrée et affiche l'entrée suivante au cycle d'horloge suivant.

Pour fournir des données à la FIFO, il suffit de mettre les données sur `din` et `wput` à 1 ; mieux vaut vérifier que full ne vaut pas 1, sinon les données peuvent être ignorées.



Voici imprimé et assemblé le 1er mouvement que j'ai finalisé.



Dès lors que j'avais un modèle fonctionnel, j'ai pu lancer le lot de 24 dont j'avais besoin.

Toutes les pièces rangées dans une boîte en carton et prêtes à être assemblées.

Attribution des bits

Au début du bloc always, il y a pas mal de manipulations de tableaux et de bits. Dans Lucid, il est facile de créer des tableaux de modules et de regrouper leurs ports en tableaux. Parfois dans notre projet, par ex. sorties de pas, nous pouvons directement remplir ces tableaux car les bits s'alignent parfaitement.

Parfois, nous devons les diviser en sous-sections. Lorsque cela se produit, il est pratique d'utiliser des boucles for. Souvenez-vous que les boucles ne peuvent pas être réalisées par le matériel et que le nombre d'itérations doit être fixé pour que la phase de «compilation» puisse les réaliser. Ces boucles ne sont qu'une convention d'écriture qui raccourcit le texte nécessaire. Par ex., la première boucle passe par 4 itérations, i allant de 0 à 3. À la première itération, la 1ère ligne sera évaluée comme suit :

```
gates.fifo_empty[0] = ani_fifos.empty[0+:12];
```

Le sélecteur de bits `[0+:12]` signifie qu'il faut sélectionner 12 bits à partir du bit 0. Ainsi, les bits 0 à 11 sont sélectionnés. À l'itération suivante, nous aurons :

```
gates.fifo_empty[1] = ani_fifos.empty[12+:12];
```

Ici, la deuxième porte d'activation reçoit les bits 12-23. Les 4 itérations seront :

```
gates.fifo_empty[0] = ani_fifos.empty[0+:12];
gates.fifo_empty[1] = ani_fifos.empty[12+:12];
gates.fifo_empty[2] = ani_fifos.empty[24+:12];
gates.fifo_empty[3] = ani_fifos.empty[36+:12];
```

Cela créera quatre fois le même circuit dans le FPGA. Je suis sûr que vous admettrez que la saisie et la maintenance du code sont lourdes. Il est très courant d'utiliser les sélecteurs start/width pour les boucles au lieu des sélecteurs start/stop. Cela vient de l'impossibilité d'utiliser les sélecteurs de bits start/stop avec des valeurs non constantes.

Le sélecteur start/width utilisé ci-dessus garantit que la largeur de la sélection est toujours de 12 bits. Il est impossible de réaliser un signal qui change de largeur dans le matériel car il est impossible de créer ou supprimer spontanément des connexions. Le `+:` sélectionne le sens croissant du sélecteur. Le `-:` fait l'inverse, c.-à-d. le sens décroissant. Cela sélectionne le bit de départ et ceux qui lui sont inférieurs. Par ex. `[11-:12]` donne le même résultat que `[0+:12]`. Tous deux sélectionnent les bits 0 à 11.

Affectation des broches

Vous vous demandez peut-être comment les signaux `step` et `dir` sont-ils affectés aux broches E/S de l'*Au*. Cette correspondance est définie dans un fichier de contraintes, *clockclock.acf*. L'extension «acf» représente les initiales d'*Alchitry Constraint File*. Ce format très simple permet de spécifier les noms des broches comme étant les broches des cartes Alchitry au lieu du FPGA. Par exemple, A2 correspond à la deuxième broche du connecteur supérieur gauche (rangée A) sur la carte *Au*. Si vous ouvrez ce fichier, vous verrez des lignes qui ressemblent à ceci :

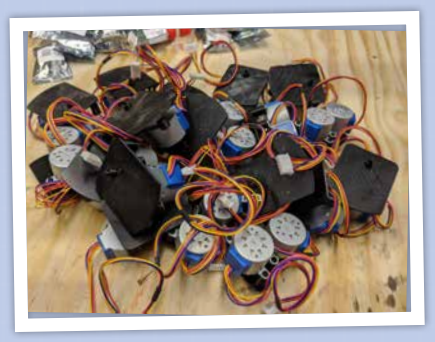
```
pin step[0] A2;
pin dir[0] A3;
```

Chaque port d'E/S doit être associé à une broche physique. Le format est le mot-clé *pin* suivi du nom du signal et de l'emplacement de la broche physique. Les mots-clés `pullup` ou `pulldown` permettent respectivement d'ajouter une résistance interne de rappel au + ou à la masse. Cependant, le `pulldown` est ignoré sur la *Cu* car le FPGA Lattice n'a pas de résistance interne de rappel à la masse.

La plupart des broches d'un FPGA sont totalement interchangeables et le brochage que j'ai utilisé pour l'horloge était arbitraire, à l'exception des signaux *Qwiic* puisqu'ils sont câblés sur le connecteur *Qwiic*. Ce qui était important pour ce projet, c'était qu'ils soient tous droits.

Installation du logiciel et programmation

Note : Cet exemple suppose que vous utilisez un EDI *Arduino* à jour. Si vous débutez avec Arduino, consultez notre tutoriel sur



24 horloges élémentaires assemblées.

L'étape suivante consistait à créer le cadre. Je suis parti de deux planches d'érable. J'ai scié la première en trois minces morceaux...

... contrecollés ensuite pour faire la façade...

Arduino + FPGA = un exemple à suivre pour de nombreux projets

l'installation de l'EDI *Arduino* [14]. Si vous installez votre 1ère bibliothèque *Arduino*, consultez notre guide d'installation [15].
Le microcontrôleur que j'ai utilisé était le *RedBoard Turbo*, mais comme susmentionné, toute carte avec connecteur *Qwiic* peut probablement être utilisée. J'ai d'abord dû installer les bibliothèques pour la carte ainsi que le *RTC RV8803* de *Qwiic* et les boutons. Les boutons ci-dessous sont des liens vers leurs tutoriels d'installation respectifs.
Le code en lui-même n'est pas trop compliqué. La structure relève de mon expérience, je l'adopte pour de nombreux projets. En gros, je construis des couches d'abstraction jusqu'à ce que j'arrive à une couche facile à utiliser pour la gestion des chiffres et la réalisation des animations.
La première couche est, ça va de soi, le FPGA. Le FPGA nous donne une interface pour faire exécuter à un moteur un certain nombre de pas avec un délai fixe entre eux. J'ai utilisé la bibliothèque *Wire* intégrée à *Arduino* pour gérer le bus *I²C*. Cela m'a permis de réaliser une fonction simple qui envoie une seule animation - la voici :

```
void sendAnimation(uint8_t id, int16_t steps, uint16_t
delay_cycles) {
  int32_t p = currentPosition[id] + steps;
  while (p < 0) p += FULL_CIRCLE;
 currentPosition[id] = p % FULL_CIRCLE;
  Wire.write(id);
  Wire.write((uint8_t)(steps >> 8));
  Wire.write((uint8_t)(steps & 0xFF));
  Wire.write((uint8_t)(delay_cycles >> 8));
  Wire.write((uint8_t)(delay_cycles & 0xFF));
}
```

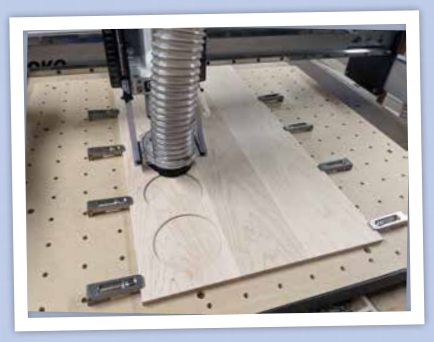
Deux particularités ici, je déclare une constante nommée `FULL_CIRCLE` pour le nombre de pas dans une rotation complète. Dans mon cas, c'est 4096. Cette constante est utilisée pour mettre à jour un tableau global des positions des moteurs. En employant toujours cette fonction pour envoyer les animations, la position des moteurs est connue.
Il n'est pas très naturel de penser un mouvement en termes de pas et de temporisation. Il est plus facile de le concevoir en termes de degrés et de durée. En d'autres termes, au lieu de penser «faire 2048 pas avec 760 cycles de tempo. entre chaque pas», il est bien plus naturel de penser «tourner de 180 ° en 4 s». J'ai donc écrit une fonction qui s'occupe de cette transposition :

```
void animate(uint8_t id, float deg, float duration) {
  float steps = deg * FULL_CIRCLE / 360.0f;
  if (steps > 32767 || steps < -32768) {
    animate(id, deg / 2, duration / 2);
    animate(id, deg / 2, duration / 2);
    return;
 }

  float cycles = constrain(duration * 390625 / abs(steps),
760, 65535);

  sendAnimation(id, (int16_t)steps, (uint16_t)cycles);
}
```

Elle commence par transformer les degrés en pas, à l'aide de la constante `FULL_CIRCLE`. Elle vérifie ensuite s'il y a trop de pas pour une seule commande d'animation et s'appelle récursivement avec la moitié de l'animation.
Les cycles de temporisation sont ensuite calculés. La valeur 390625 est le nombre de cycles en une seconde (100 000 000 / 256 = 390 625). Les cycles doivent être cantonnés à la plage de 760 à 65535. Le minimum de 760 a été trouvé empiriquement en testant la vitesse à laquelle les moteurs peuvent tourner sans sauter de pas. Cela correspond à 8 s par tour.

… que j'ai alors dressée puis j'ai découpé le pourtour avec ma CNC.

L'horloge est trop grande pour mon Shapeoko XXL, j'ai donc dû procéder en 2 fois pour chaque opération. Une fois une face terminée, j'ai légèrement fraisé l'autre face pour un dressage parfait.



La planche était alors prête pour le fraisage des logements des horloges.

La limite supérieure est très élevée et ne devrait jamais être atteinte. Il faudrait 687 s pour faire une rotation complète. Le projet actuel ne peut pas aller plus lentement que cela. Au besoin, il faudrait changer le prédimensionnement du FPGA ou la longueur du cycle de temporisation.
Il me fallait enfin une fonction permettant de dire au moteur de se déplacer jusqu'à une position donnée. La `CurrentPosition` permet à cette fonction de calculer la rotation minimale nécessaire pour l'atteindre. C'est utile pour afficher l'heure réelle, car je peux l'appeler quelles que soient les positions réelles des aiguilles, sans faire référence à leur position en cours.

```
void moveTo(uint8_t id, float pos, float duration) {
  float curDeg = (float)currentPosition[id] * 360.0f /
FULL_CIRCLE;
  float angle = pos - curDeg ;

  if (angle > 180)
    angle = angle - 360;
  if (angle < -180)
    angle = 360 + angle;

 animate(id, angle, duration);
}
```

Cela commence par le calcul de l'angle auquel l'aiguille se trouve à cet instant. Le déplacement angulaire s'obtient en soustrayant de l'angle souhaité l'angle calculé ci-avant. Mais cet angle pourrait ne pas donner le plus court des deux chemins ; les deux instructions if se chargent de le vérifier pour choisir le plus court. Par ex., si la différence entre les angles est de +270 °, il serait préférable de se déplacer de -90 °.

Pour afficher l'heure réelle, il me fallait une carte pour tous les chiffres. Il a suffi de noter les angles de chaque aiguille tels qu'ils forment les chiffres voulus . J'ai saisi le tout dans un tableau 2D (**listage 4**) utilisable pour retrouver n'importe quel chiffre. La RTC me donne l'heure dont il suffit d'afficher les chiffres :

```
void showTime() {
  uint8_t digits[4];
  digits[0] = rtc.getMinutes() % 10;
  digits[1] = rtc.getMinutes() / 10;
 digits[2] = rtc.getHours() % 10;
  digits[3] = rtc.getHours() / 10;
  for (uint8_t d = 0; d < 4; d++) {
    Wire.beginTransmission(0x50);
    for (uint8_t m = 0; m < 12; m++) {
      moveTo(m + 12 * d, digitAngles[digits[d]][m], 4.0f);
  }
    Wire.endTransmission();
  }
}
```

Il convient de mentionner que mon code suppose que les aiguilles commencent à la position 12h00. C'est le repère 0 ° pour toutes les positions.
Dans la fonction `loop()` *Arduino*, j'ai inclus du code qui permet de lire la RTC toutes les 100 ms et de mettre l'heure à jour. Au changement d'heure entière, je lance également une animation. J'en ai écrit trois simples dont l'une choisie au hasard s'exécute avant de fixer la nouvelle heure.
À l'intérieur de `loop()`, je vérifie également l'état des 4 boutons. Mon plan était d'utiliser l'interface FIFO du bouton *Qwiic* pour garder la trace de chaque pression, mais je suis tombé sur un os. J'ai fini par suivre moi-même l'état et la fonction `isPressed()` n'a servi qu'à vérifier leur état instantané.
Lorsqu'un bouton donné est enfoncé, je mets l'heure à jour. Les 4 boutons permettent de régler indépendamment minutes et

Une fois cette plaque de façade finie, elle s'est révélée un peu plus fine que prévu après le dressage. Le fond des logements ne faisait plus qu'1,5 mm d'épaisseur. Cela s'est révélé suffisamment solide.

J'ai ensuite construit un cadre et je l'ai collé…

… pour obtenir un solide coffret en bois.

heures. J'ai ajouté une fonction qui permet, avec un appui simultané sur les 2 boutons *Hour Up* et *Hour Down*, de faire indiquer 12h00 aux 24 horloges. C'est très utile s'il faut éteindre l'horloge ou reprogrammer l'Arduino, cela évite de réajuster chaque aiguille à la main.
Voilà qui résume bien le projet. Partez d'une interface simple et développez-la jusqu'à joindre l'utile à l'agréable.

Conclusion

Ce projet a demandé beaucoup plus de travail que je pensais au départ. La majeure partie du temps a été consacrée à la réalisation matérielle et au câblage. La conception et la mise au point un mouvement fonctionnel et fiable m'ont également pris un temps *certain*. La conception des parties FPGA et *Arduino* n'a souffert d'aucun contretemps notable.
L'utilisation du bus *Qwiic* de l'*Au* pour communiquer avec le microcontrôleur est sans doute une application des plus intéressantes. N'ayant jamais utilisé l'*I²C* sur un *Arduino* auparavant, j'ai été agréablement surpris par la facilité d'installation côté *Arduino*.
Si quelqu'un fabriquait une autre *ClockClock*, il y aurait plusieurs points à améliorer.

- L'horloge est assez bruyante. Pris individuellement, chaque moteur est assez silencieux, mais le fait de les coller ensemble sur un mince morceau de bois amplifie notablement le bruit. J'aurais dû prévoir un moyen d'amortir le bruit lors du montage des moteurs. Une colle à base de caoutchouc pourrait être une solution. De plus, le bois n'a que 1,5 mm d'épaisseur sur la majeure partie de la façade, laquelle fait donc caisse de résonnance. Je pourrais essayer de verser du caoutchouc liquide sur la face arrière de la planche pour essayer d'amortir le bruit.
- L'autre problème majeur tient au jeu du réducteur interne des moteurs pas à pas . En pratique, selon le sens de déplacement, l'aiguille peut ou non se trouver là où elle est censée être. À cause du jeu, l'écart semble faire quelques degrés lors des inversions de sens. Ce n'est pas la fin du monde, mais une fois qu'on s'en est aperçu, on ne voit rien d'autre. Je pourrais sans doute compenser cela par programmation, mais le jeu varie d'un moteur à l'autre et il faudrait un réglage individuel de la compensation. On pourrait aussi y remédier avec des moteurs sans engrenage.

J'espère que ce projet constitue une bonne démonstration des possibilités des FPGA et qu'il allumera (spark) en vous l'envie de prendre du plaisir (fun) à vous lancer dans votre propre *ClockClock* !

(200676-01 VF : Yves Georges)

Espace Elektor en ligne pour les liens internet :
www.elektormagazine.fr/esfe-en-clockclock

La conception des parties FPGA et Arduino de ce projet n'a souffert d'aucun contretemps notable. C'est la réalisation matérielle, la mise au point mécanique et le câblage qui sont chronophages.

Pour les aiguilles, j'ai choisi le padouk, un beau bois rouge qui brunit avec le temps. C'est aussi le bois des tiroirs de ma cuisine où l'horloge sera accrochée et j'ai pensé que ça ferait bien s'ils étaient assortis. J'ai usiné les aiguilles dans une planche de 3 mm fraisée jusqu'à une épaisseur de 2 mm.

Avec les aiguilles terminées…

… J'ai collé toutes les sous-horloges au boîtier de l'horloge. Quatre d'entre elles sont orientées curieusement pour dégager la place de l'alimentation. C'est une alimentation 12 V 6 A suffisante pour tous les moteurs.

Accesssoires

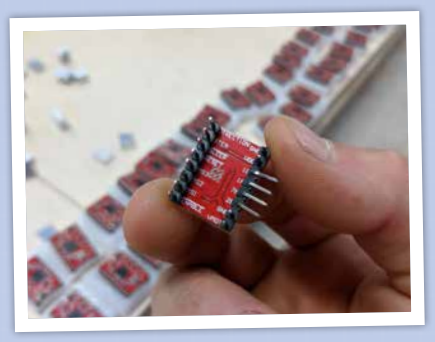
Si vous cherchez les accessoires mentionnés dans cet article, vous en trouverez certains chez SparkFun et Elektor.

- (2) Hook-Up Wire - Assortment (Solid Core, 22 AWG); PRT-11367
- (2) Qwiic Cable - 100 mm; PRT-14427
- (3) Qwiic Cable - 50 mm; PRT-14426
- SparkFun Real Time Clock Module - RV-8803 (Qwiic); BOB-16281
- (4) SparkFun Qwiic Button - Red LED; BOB-15932
- Alchitry Br Prototype Element Board; DEV-16524
- Alchitry Au FPGA Development Board (Xilinx Artix 7); DEV-16527
- SparkFun RedBoard Turbo - SAMD21 Development Board; DEV-14812

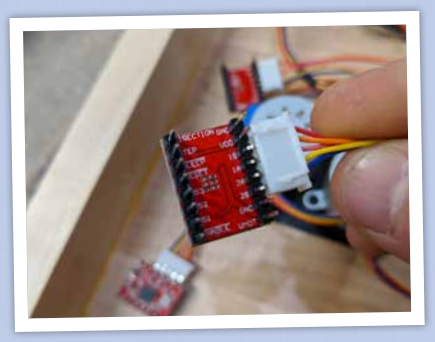
www.elektormagazine.fr/esfe-en-ClockClock1

LIENS

[1] ClockClock by Humans since 1982: https://clockclock.com
[2] Alchitry Au IO pins: https://www.elektormagazine.fr/esfe-en-ClockClock1
[3] Valve Gear Stepper Motor : https://www.amazon.com/gp/product/B01J3KV3B2
[4] StepStick Stepper Motor Diver Module with Heat Sink: https://www.amazon.com/gp/product/B01FFGAKK8
[5] UBEC Adjustable BEC UBEC 2-6S for Quadcopter RC Drone : https://www.amazon.com/gp/product/B07PLSYX9G
[6] Enclosed AC-DC Switching Power Supply : https://www.amazon.com/gp/product/B07Z55FCQQ
[7] Programming an FPGA: https://bit.ly/3nplgif
[8] How Does an FPGA Work? : https://bit.ly/3r42FdG
[9] Getting Fancy with PWM: https://bit.ly/2IUDmcZ
[10] CAD file (Fusion 360) : https://bit.ly/3mpXSjo
[11] ClockClock Alchitry fille (FPGA) : https://bit.ly/2K90HrW
[12] ClockClock Arduino code (zip) : https://bit.ly/3moeXdv
[13] 28BYJ-48 stepper motors: https://www.amazon.com/gp/product/B01J3KV3B2
[14] Arduino IDE installation: https://bit.ly/2Lz6OWM
[15] Installing an Arduino library: https://bit.ly/37owopV
[16] Learning FPGAs: Digital Design for Beginners with Mojo and Lucid HDL: https://amzn.to/38aexlN



J'ai utilisé des pilotes de moteur pas à pas génériques A4988 (www.amazon.com/gp/product/B01FFGAKK8). J'ai coudé les broches de commande des moteurs…

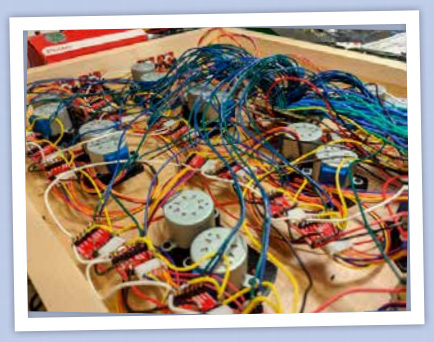


… et pu y brancher directement les connecteurs des moteurs après interversion de deux des fils.

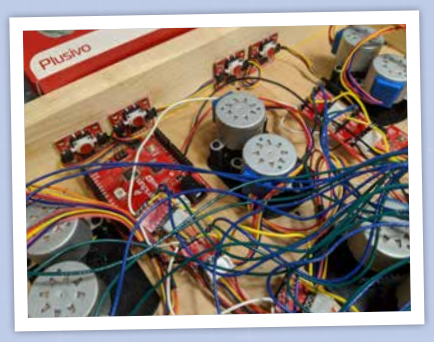
Il ne restait plus qu'à mettre sous tension. Sur cette photo, deux mouvements sont absents car il me manquait encore 4 moteurs.

Listage 1.

```
module animator (
    input clk,  // horloge
    input rst,  // reset
    signed input stepCount[16], // peut être négatif pour indiquer le sens
    input delayCycles[16],      // cycles entre chaque pas
    input newAnimation,         // drapeau pour nouvelle animation
    output busy,                // signale que l'animateur est occupé et n'accepte pas les animations
    output step,                // signal de pas pour le driver de moteur
    output direction            // signal de sens pour le driver de moteur
  ) {

  .clk(clk) {
    // Le driver exige que l'impulsion de „pas“ dure au moins 1 µs, avec 2 µs c'est bien
    pulse_extender stepExt(#MIN_PULSE_TIME(2000));

    dff dirCt[8];      // compteur pour la tempo après inversion de sens. Il faut 200 ns
    dff counter[16+8]; // compteur de tempo entre les pas. Le +8 est le prédiviseur

    .rst(rst) {
      fsm state = ;
      dff dir;         // sens de rotation du moteur mémorisé
      dff delayCt[16]; // valeur de tempo mémorisée
      dff steps[16];   // nombre de pas mémorisé (valeur absolue)
    }
  }

  always {
    busy = state.q != state.IDLE; // occupé quand il n'est pas inactif
    step = stepExt.out;           // la sortie step est l'impulsion étendue

    stepExt.in = 0;               // valeur par défaut du nouveau pas
    direction = dir.q;            // sortie du sens mémorisé

    case (state.q) {
      state.IDLE:
        if (newAnimation && stepCount != 0) {
                                  // si nouvelle animation avec des pas (sauter les animations avec 0 pas)
```



Chaque pilote devait être relié à l'alimentation 12 V pour les moteurs et 3,3 V pour la logique de commande. Sur chaque groupe de six constituant un chiffre, j'ai relié entre elles les broches de validation puis les ai reliées à l'*Alchitry Au*. Il fallait également raccorder à l'*Au* les signaux de sens et de pas de chaque pilote. Quel embrouillamini !



Il me fallait un moyen de régler l'heure, alors j'ai ajouté quatre boutons *Qwiic*. J'en avais un peu assez du câblage, et l'utilisation des connecteurs *Qwiic* a bien facilité les choses. Les deux du haut font *Heure +/-* et les deux du bas *Minute +/-*. Après les boutons, il y a une RTC (horloge en temps réel) *RV-8803* également connectée au bus *Qwiic*. La RTC étant alimentée par une batterie, je n'ai pas besoin de régler l'heure chaque fois que je reprogramme ou débranche la carte. Enfin, j'ai connecté l'*Alchitry Au*. Elle doit se trouver en bout de chaîne puisqu'elle n'a qu'un seul connecteur *Qwiic*. J'ai également retiré le fil d'alimentation du câble *Qwiic* pour que le régulateur 3,3 V de l'*Alchitry Au* n'entre pas en conflit avec celui de la

```
          state.d = state.DIR_WAIT;   // passer à l'état suivant
          dir.d = stepCount[stepCount.WIDTH-1];
                                      // le sens est le signe de l'entrée de pas (0 = positif, 1 = négatif)
          steps.d = stepCount[stepCount.WIDTH-1] ? -stepCount : stepCount;
                                      // mémorise la valeur absolue de stepCount
          delayCt.d = delayCycles;    // mémorise le nombre de cycles de temporisation
        }
      state.DIR_WAIT:
        dirCt.d = dirCt.q + 1;        // attendre la sortie du sens après son inversion
        if (&dirCt.q) {               // si tempo terminée
          state.d = state.STEP;       // passer à l'état rotation pas à pas
        }
      state.STEP:
        counter.d = counter.q + 1;    // incrémenter le compteur de tempo des pas
        if (counter.q[counter.WIDTH-1-:16] == delayCt.q) { // si le compteur a atteint la tempo finale
          counter.d = 0;              // réinit. le compteur
          stepExt.in = 1;             // envoyer une impulsion
          steps.d = steps.q - 1;      // décrémenter le nombre de pas restant
          if (steps.q == 1) {         // s'il n'en reste plus
            state.d = state.IDLE;     // retour à l'état d'inactivité
          }
        }
    }
  }
}
```

Listage 2.

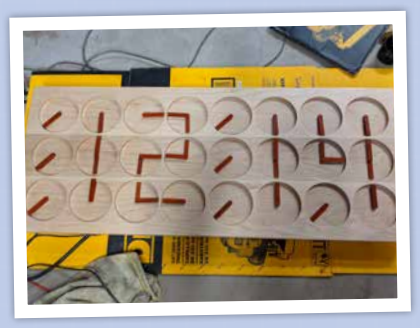
```
module enable_gate (
    input clk,  // horloge
    input rst,  // réinit.reset
    output new_animation[12], // sortie vers les animateurs
    input fifo_empty[12],     // entrée venant des fifos (animations en file d'attente)
    input animator_busy[12],  // entrée venant des animateurs
    output enable             // sortie vers les drivers des moteurs
  ) {
```

RedBoard Turbo. *** Le microcontrôleur est une carte *RedBoard Turbo*. Je l'ai utilisée parce qu'elle avait un connecteur *Qwiic* et que je l'avais sous la main. Tout microcontrôleur avec un connecteur *Qwiic* peut être utilisé. La puissance de calcul nécessaire est minime. *** L'*Alchitry Au* et la *RedBoard Turbo* nécessitent toutes deux du 5 V. J'ai utilisé un petit régulateur abaisseur (12 V en 5 V) commun en radiocommande et débitant jusqu'à 2 A. *** J'ai câblé les drivers pas à pas pour qu'ils utilisent le demi-pas. Au départ, ma configuration utilisait un micro-pas de 1/16e, mais les moteurs gémissaient s'ils n'étaient pas sur un demi-pas ou un pas entier. Une telle résolution était un peu superflue.

Le brochage des 102 fils qui vont dans le FPGA n'a pas vraiment d'importance. Il faut seulement noter comment vous les avez câblés. Le brochage réel est défini par le fichier de contraintes du projet FPGA.

Une fois le câblage fait, j'y ai mis un peu d'ordre…

```
  .clk(clk) {
    .rst(rst) {
      dff onCtr[22];   // compteur pour être sûr que les moteurs sont bien en marche (22 bits ~ 42 ms)
      dff offCtr[22];  // compteur de maintien en marche après la fin des animateurs (22 bits ~ 42 ms)
    }
  }

  sig running; // valeur utilisée pour savoir quand les moteurs doivent être en marche

  always {
    // exécuter si des animations sont en attente ou en cours d'exécution
    running = |(animator_busy | ~fifo_empty);

    // le drapeau enable est activé si l'option onCtr est activée ou n'est pas 0 (réinitialisé après un
dépasst de offCtr)
    enable = running || (onCtr.q != 0);

    // ne transmettre le drapeau new_animation que lorsque onCtr est plein
    new_animation = ~fifo_empty & 12x{&onCtr.q};

    if (running) {
      offCtr.d = 0;              // réinit. le compteur d'arrêt moteur
      if (!&onCtr.q) {           // si non plein
        onCtr.d = onCtr.q + 1;   // incrément onCtr
      }
    } else {                     // à l'arrêt
      if (!&offCtr.q) {          // si offCtr non plein
        offCtr.d = offCtr.q + 1; // incrémente offCtr
      } else {                   // si offCtr plein
        onCtr.d = 0;             // réinit. onCtr
      }
    }
  }
}
```



... et placé les aiguilles sur l'horloge.

Vous savez l'essentiel pour la construction physique. Le câblage est fastidieux, mais pas compliqué.

Listage 3.

```
module au_top (
    input clk,          // horloge 100 MHz
    input rst_n,        // bouton de réinitialisation (actif à 0)
    output led [8],     // 8 LED contrôlables par l'utilisateur
    input usb_rx,       // entrée USB->Série
    output usb_tx,      // sortie USB->Série
    output step[48],    // sortie des pas vers les moteurs
    output dir[48],     // sortie du sens de rotation vers les moteurs
    output enable[4],   // sortie activation vers les moteurs (une par digit)
    inout sda,          // SDA Qwiic
    input scl           // SCL Qwiic
  ) {

  sig rst;              // signal de réinit.

  .clk(clk) {
    // Le conditionneur de réinit. est utilisé pour synchroniser le signal de réinit. vers
    // l'horloge FPGA.

Cela garantit que le FPGA entier sort de la réinit. en même temps.
    reset_conditioner reset_cond;

    dff ani_id[6];            // octet ID mémorisé pour le moteur
    signed dff ani_steps[16]; // nombre de pas mémorisé
    dff ani_delay[16];        // compte de temporisation mémoriséla tempo

    dff byteCt;               // drapeau d'octet pour les nombres 16 bits

    .rst(rst) {
      i2c_peripheral qwiic (.sda(sda), .scl(scl)); // module périphérique i2c pour l'interface qwiic

      dff ledReg[8]; // registre de mémorisation des valeurs des LED (utile pour les tests qwiic)

      fsm state = ;

      animator animators[48]; // il faut 48 animateurs (un par moteur)

      // il faut une fifo par animateur, 32 bits de large (16 bits nbre de pas + 16 bits tempo)
      // la profondeur de 128 est excessive et 16 serait probablement suffisant pour
      // l'usage actuel.
      fifo ani_fifos[48] (#SIZE(32), #DEPTH(128));

      enable_gate gates[4]; // modules de contrôle des signaux d'activation (un par chiffre)
    }
  }

  var i;

  always {
    reset_cond.in = ~rst_n; // entre le signal de réinitialisation brut inversé
    rst = reset_cond.out;   // reset conditionné

    led = ledReg.q;         // envoie ledReg sur les leds

    usb_tx = usb_rx;        // renvoie l'écho des données série reçues

    // le ~ inverse le sens du moteur pour que les pas positifs se fassent dans le sens horaire
    // 4hA = 1010, donc dans 48hAAAAAAAAAAAAAA, 1 bit sur 2 est à 1, de sorte que les
    // moteurs feront tourner les aiguilles des heures et des minutes dans le même sens
    dir = animators.direction ^ ~48hAAAAAAAAAAAA;
    step = animators.step;
```

```
enable = ~gates.enable; // l'activation des contrôleurs est active à 0, donc inversion des bits

qwiic.tx_data = 8bx; // ce projet est „en écriture seule“ et n'envoie jamais de données au microcontrôleur
qwiic.tx_enable = 0; // ne jamais envoyer de données

// groupes combinés de 12 moteurs pour les 4 portes de validation
for (i = 0; i < 4; i++) {
  gates.fifo_empty[i] = ani_fifos.empty[i*12+:12];
  gates.animator_busy[i] = animators.busy[i*12+:12];
  animators.newAnimation[i*12+:12] = gates.new_animation[i];
  // ne supprime une valeur de la fifo que si la porte passe le drapeau new_animation
  // et que l'animateur n'est pas occupé
  ani_fifos.rget[i*12+:12] = gates.new_animation[i] & ~animators.busy[i*12+:12];
}

// pour chaque moteur, diviser la sortie fifo vers les signaux de l'animateur
for (i = 0; i < 48; i++) {
  animators.stepCount[i] = ani_fifos.dout[i][15:0];
  animators.delayCycles[i] = ani_fifos.dout[i][31:16];
}

// pas de nouvelles animations par défaut
ani_fifos.wput = 48b0;

// toujours entrer la tempo. et les pas mémorisés
// Concatène les 2 valeurs de 16 bits en 32 bits, et les range dans un tableau 1x32,
// et enfin le duplique 48 fois pour former un tableau 48x32
// cela ne fait qu'alimenter chacune des 48 fifos avec les mêmes 32 bits à
ani_fifos.din = 48x{}};

case (state.q) {
  state.IDLE:
    byteCt.d = 0;
    if (qwiic.rx_valid) {          // nouvelles données
      case (qwiic.rx_data) {       // cas selon la valeur de l'adresse I2C
        8hFF: state.d = state.LED; // faire „addresse“ FF pour les LEDs pour le test
        default:
          ani_id.d = qwiic.rx_data[5:0];  // par défaut „adresse“ comme identifiant du moteur
          state.d = state.ANIMATION_STEPS;
      }
    }
  state.LED:
    if (qwiic.rx_valid) {        // si nouvelles données
      state.d = state.IDLE;      // retour à l'état d'inactivité
      ledReg.d = qwiic.rx_data;  // afficher la valeur sur les LED
    }
  state.ANIMATION_STEPS:
    if (qwiic.rx_valid) {                // si nouvelles données
      ani_steps.d = c;  // mémorise l'octet et décale l'ancien octet
      byteCt.d = ~byteCt.q;              // inverse le compteur d'octet
      if (byteCt.q == 1) {               // si deuxième octet
        state.d = state.ANIMATION_DELAY;  // passer à l'état de capture de la tempo
      }
    }
  state.ANIMATION_DELAY:
    if (qwiic.rx_valid) {                // si nouvelles données
      ani_delay.d = c;  // sauvegarde l'octet et décale l'ancien octet
      byteCt.d = ~byteCt.q;              // inverse le compteur d'octet
      if (byteCt.q == 1) {               // si deuxième octet
        state.d = state.ANIMATION_PUT;    // passer à l'état remplissage fifo
      }
    }
  state.ANIMATION_PUT:
    state.d = state.IDLE;                // retour à l'état d'inactivité
```

```
        ani_fifos.wput[ani_id.q] = 1;         // mettre la nouvelle animation dans la fifo correcte
    }

    if (qwiic.stop) {           // si une condition d'arrêt I2C est détectée
      state.d = state.IDLE;     // retour à l'état d'inactivité
    }
  }
}
```

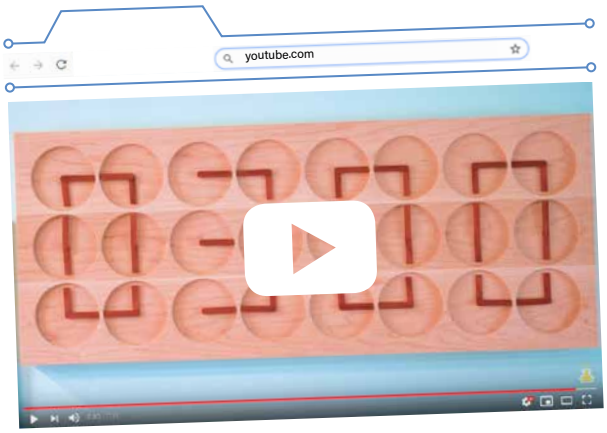
Listage 4.

```
const float digitAngles[10][12] = {{270, 180, 0, 180, 270, 0, 90, 180, 0, 180, 90, 0}, // 0
  {180, 180, 0, 180, 0, 0, 225, 225, 225, 225, 225, 225}, // 1
  {270, 180, 270, 0, 270, 270, 90, 90, 90, 180, 90, 0}, // 2
  {270, 180, 0, 180, 270, 0, 90, 90, 90, 90, 90, 90}, // 3
  {180, 180, 0, 180, 0, 0, 180, 180, 90, 0, 225, 225}, // 4
  {270, 270, 270, 180, 270, 0, 90, 180, 90, 0, 90, 90}, // 5
  {270, 270, 270, 180, 270, 0, 90, 180, 0, 180, 90, 0}, // 6
  {270, 180, 0, 180, 0, 0, 90, 90, 225, 225, 225, 225}, // 7
  {270, 180, 270, 0, 0, 270, 90, 180, 90, 0, 0, 90}, // 8
  {270, 180, 0, 180, 0, 0, 90, 180, 90, 0, 225, 225} // 9
};
```

Vidéo de démonstration

Vous pouvez voir fonctionner le proto de Justin sur YouTube :

https://youtu.be/rNJlQ4Fa9mQ



Un coup ça marche, un coup ça marche pas

Si vous avez besoin d'aide ou d'informations, veuillez parcourir les forums d'Alchitry sur https://forum.alchitry.com. Vous y trouverez probablement la réponse à vos questions ou les personnes susceptibles d'y répondre.

Pour approfondir

1. Fichiers de production :

- CAD File (Fusion 360) [10]
- Alchitry (FPGA) [11]
- Arduino Code (ZIP) [12]

2. Le site web d'Alchitry (https://alchitry.com) propose d'autres ressources intéressantes, notamment des tutoriels et projets, un forum, le schéma Alchitry Au (PDF), le schéma Alchitry Cu (PDF) et le guide d'utilisation de Xilinx Artix 7.

3. Pour approfondir vos connaissances des FPGA et de Lucid, consultez la page *Learning FPGAs: Digital Design for Beginners with Mojo and Lucid HDL* par Justin Rajewski [16]. C'est une excellente ressource pour comprendre et finalement concevoir vos propres applications FPGA.