ÉLECTRONIQUE | EMBARQUÉ | RADIO | IOT

HACKABLE

L'EMBARQUÉ À SA SOURCE

N° 55
JUILLET / AOÛT 2024

FRANCE MÉTRO. : 14,90 €
BELUX : 15,90 € - CH : 23,90 CHF ESP/IT/PORT-CONT : 14,90 €
DOM/S : 14,90 € - TUN : 35,60 TND - MAR : 165 MAD - CAN : 24,99 $CAD

L 19338 - 55 - F: 14,90 € - RD

CPPAP : K92470

HACK / OSCILLOSCOPE

Vous n'avez que deux mains pour faire vos mesures ? Ajoutez une pédale USB à votre oscilloscope ! p.18

SÉCURITÉ / PROTOCOLE

Créez un environnement pour tester la sécurité des systèmes industriels basés sur Modbus p.34

Circuits programmables / VHDL / Z80

Toujours du mal à aller au-delà de la simple LED qui clignote ?

MON PREMIER (VRAI) PROJET FPGA !

p.54

- Choisir sa plateforme
- Réunir CPU, RAM, ROM et UART
- Créer son ordinateur 8 bits

ALIMENTATION / USB

ALIENTEK DP100 : Prise en main et exploration d'une minuscule alimentation de laboratoire p.04

RÉTRO / MODEM

RS-232, carte FXS, RTC, PPP, Asterisk, Amstrad CPC 464... Et si on surfait comme en 1989 ? p.112

ICESTICK / OPTIMISATION

Apprenez à augmenter les performances de vos circuits écrits en Chisel sur plateforme ICE40 p.88

connect.ed-diamond.com

ÉDITION #4

OPEN SOURCEZ VOS SOLUTIONS IT

OPEN SOURCE EXPERIENCE

PARIS - 04 & 05 DÉCEMBRE 2024 - - PALAIS DES CONGRÈS

90 EXPOSANTS **100 CONFÉRENCES** **125 SPEAKERS**

www.opensource-experience.com

Suivez-nous #OSXP2024

Un événement Systematic Paris Region Deep Tech Ecosystem organisé par infoprodigital TRADE SHOWS

NOUVEAU CETTE ANNÉE ! Aux mêmes dates et lieu que

DEVOPS REX
LA CONFÉRENCE DEVOPS FRANCOPHONE
100% retour d'expérience

ÉDITO

Quelque chose est sur le point de changer dans l'embarqué… et pas seulement.

La sécurité des systèmes est plus que jamais sur le devant de la scène. L'industrie entière s'est enfin rendu compte des répercussions potentiellement catastrophiques de failles de sécurité au plus bas niveau et redouble d'efforts pour « régler » le problème. Mais on a beau tenter de « sécuriser » les systèmes, les applications, les services et même les langages, si une faiblesse existe, héritée de plusieurs dizaines d'années d'histoire de l'informatique moderne, elle sera forcément exploitée. Cette faiblesse est la manière dont l'accès à la mémoire est géré par les processeurs et la corruption de cette mémoire, d'une façon ou d'une autre, représente statistiquement la principale cause des failles de sécurité.

Pour régler le problème, un nouveau paradigme a vu le jour et est sur le point de se généraliser, en particulier dans l'embarqué (dans un premier temps) : les *capabilities* et plus exactement l'adressage mémoire basée sur les *capabilities*. Le concept n'est pas nouveau, puisqu'il s'agit simplement d'associer une métadonnée à un objet (au sens large du terme), pour conférer ou non une « capacité » (d'où le nom) à manipuler cet objet. Ce qui est (presque) nouveau, en revanche, c'est d'intégrer ce mécanisme dans le processeur lui-même sous la forme d'une gestion mémoire étendue et d'instructions spécifiques. « Presque », parce que le System/38 d'IBM (1978) utilisait cette technologie, mais depuis, le mécanisme de pagination mémoire a été universellement adopté.

Le projet vers lequel convergent les recherches depuis quelque temps est celui de l'université de Cambridge : CHERI (pour *Capability Hardware Enhanced Risc Instructions*). Les constructeurs (ARM en tête avec son projet Morello, mais RISC-V n'est pas en reste), les éditeurs de logiciels, ainsi que les projets *open source* (Morello Linux, CheriBSD, LLVM, FreeRTOS, etc.) forcent la marche pour nous rapprocher d'un premier produit fini.

Difficile de prédire quand cette révolution aura lieu, et débarquera dans nos smartphones et nos SBC, mais elle est en route, et ça changera énormément de choses…

Denis Bodor

Hackable Magazine
est édité par Les Éditions Diamond

BP 20142 - 67602 SELESTAT CEDEX - France
E-mail : lecteurs@hackable.fr -
Service commercial : cial@ed-diamond.com
Sites : hackable.fr - ed-diamond.com
Directeur de publication : Arnaud Metzler
Rédacteur en chef : Denis Bodor
Réalisation graphique : Kathrin Scali
Régie publicitaire :
Valérie Fréchard - Tél. : 03 67 10 00 27
Service abonnement : Les Éditions Diamond
BP 20142 - 67602 SELESTAT CEDEX, France,
Tél. : 03 67 10 00 20
Impression : Westermann Druck | PVA, Braunschweig, Allemagne
Distribution France :
(uniquement pour les dépositaires de presse)
MLP Réassort : Plate-forme de Saint-Barthélemy-d'Anjou. Tél. : 02 41 27 53 12

Plate-forme de Saint-Quentin-Fallavier.
Tél. : 04 74 82 63 04
Service des ventes :
Abomarque - Tél. : 06 15 46 15 88
IMPRIMÉ en Allemagne - PRINTED in Germany
Dépôt légal : À parution
N° ISSN : 2427-4631
CPPAP : K92470
Périodicité : bimestriel - Prix de vente : 14,90 €

POUR DEVENIR AUTEUR

Contactez :
contrib@hackable.fr

Consultez :

Suivez-nous sur Twitter
@hackablemag

Alsace

SOMMAIRE

À PROPOS DE HACKABLE…

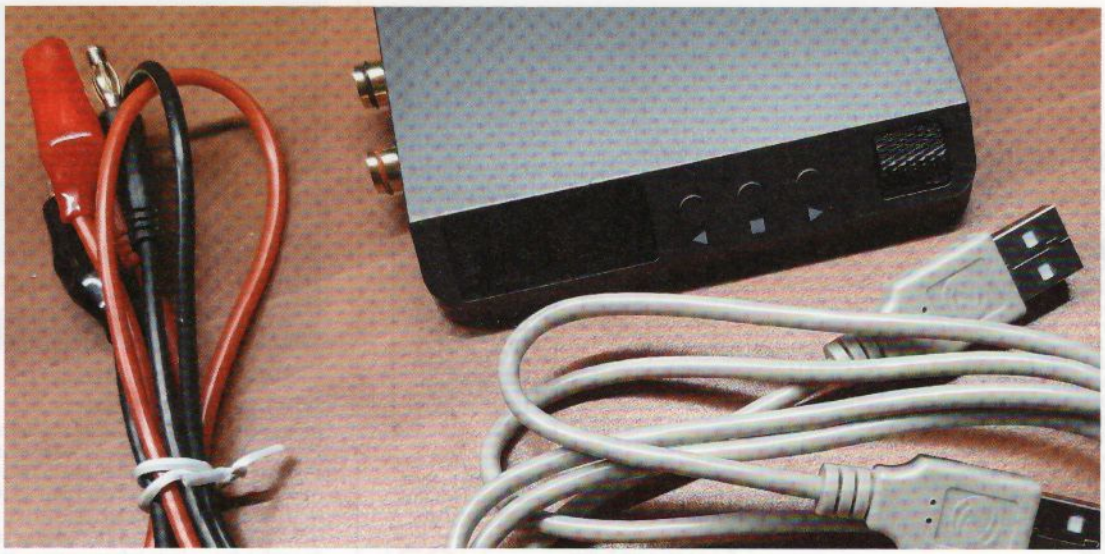
HACKS, HACKERS & HACKABLE

Ce magazine ne traite pas de piratage. Un **hack** est une solution rapide et bricolée pour régler un problème, tantôt élégante, tantôt brouillonne, mais systématiquement créative. Les personnes utilisant ce type de techniques sont appelées **hackers**, quel que soit le domaine technologique. C'est un abus de langage médiatisé que de confondre « pirate informatique » et « hacker ». Le nom de ce magazine a été choisi pour refléter cette notion de **bidouillage créatif** sur la base d'un terme utilisé dans sa définition légitime, véritable et historique.

ALIMENTATION DE LABORATOIRE ALIENTEK DP100 : PETITE, MAIS COSTAUD

Denis Bodor

L'alimentation de laboratoire fait partie des outils classiques lorsqu'on fait de l'électronique et de l'embarqué. Ceci prend généralement la forme d'un équipement volumineux destiné à prendre place sur un bench, entre la station de soudure et l'oscilloscope. Cependant, les mœurs ont sensiblement changé et souvent quelque chose de plus compact, pouvant prendre place sur un bureau à gauche du clavier est non seulement suffisant, mais plus adapté et ergonomique pour de « petits travaux ». Dans cette catégorie « outil miniature », je vous présente donc l'ALIENTEK DP100 USB-C...



– Alimentation de laboratoire ALIENTEK DP100 : petite, mais costaud –

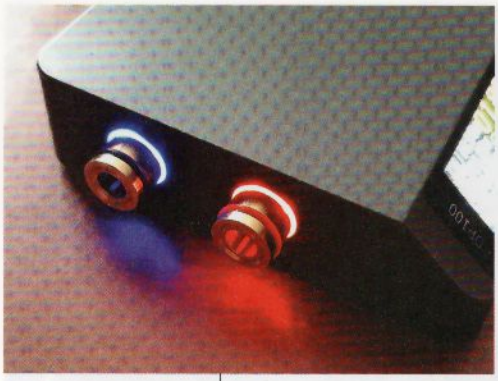
Imaginer une alimentation de labo piochant son courant via un port USB-C peut faire doucement rigoler, mais ne vous y trompez pas, l'USB de pépé, c'est de l'histoire ancienne. Le « U » de USB, et en particulier d'USB-C, est aujourd'hui véritablement « *Universal* », tant ce connecteur est utilisé, aussi bien pour le transfert de données haut débit que pour l'alimentation et la charge. Ceci au point que même certains obsédés de la pomme californiens se sont vus contraints (merci, l'Europe) d'opter pour ce connecteur en lieu et place de leur propre création (« *Lightning* » se traduisant, selon mes sources, en « lol, donne-nous tes sous » dans un ancien dialecte mérovingien du sud-ouest de l'Ouzbékistan). On trouve de l'USB-C sur les *laptops*, les smartphones, les réglettes lumineuses, les multimètres, des disques externes, les clés de sécurité, etc. Et ce n'est pas que pour l'échange de données…

L'USB-C, ou plus exactement l'USB Type-C, est un simple connecteur, rien de plus et il doit être vu, dans une certaine mesure, comme étant détaché de l'USB, l'USB 3, 3.1, etc. En effet, ce connecteur de 24 broches permet le transport d'informations avec bien d'autres protocoles de transmission que l'USB, parmi lesquels DisplayPort, Thunderbolt, PCIe, HDMI… Et ce n'est pas tout, la spécification *USB Power Delivery* (ou USB-PD) décrit également comment, et avec quelles caractéristiques, du courant peut être délivré à un périphérique, et ce avec une tension allant bien au-delà des 5 volts habituels que l'ont connaît pour l'USB « classique ».

Le DP100 d'ALIENTEK est compact, très compact. Ceci s'avère certes pratique pour un usage nomade, mais uniquement à condition d'avoir des petits doigts et de bons yeux.

Avec une alimentation USB-PD et un périphérique compatible, le courant et la tension sont négociés entre les deux entités, permettant ainsi une charge rapide (pour les smartphones et *laptops*, par exemple). Ainsi, un unique bloc d'alimentation, comme mon UGREEN Nexode 65 W (~40 € sur Amazon), sera capable de fournir non seulement jusqu'à 5 A en 5 V pour un accu d'appoint USB ou un montage quelconque par exemple, mais également jusqu'à 3,25 A en 20 V pour mon portable Lenovo E480, remplaçant l'encombrant adaptateur d'origine livré avec l'ordinateur.

Pour éviter toute confusion, sur ce point du moins, une certification a été mise en place afin de permettre à l'utilisateur d'identifier clairement quels chargeurs et périphériques sont capables de supporter la gamme de puissances (15 W, 27 W, 45 W, 60 W, 100 W, 140 W, 180 W, 240 W) et de tensions (5 V, 9 V, 15 V, 20 V, 28 V, 36 V, 48 V) : c'est le fameux « *Fast Charge* » ou « *Certified USB Fast Charger* », accompagné d'un logo dédié, que l'on retrouve sur les différents équipements concernés.



Il faut l'avouer, même si l'utilité de cette fonctionnalité est relativement limitée (soyons honnêtes), les connecteurs fiche « banane » s'éclairant lorsque la sortie est active sont du plus bel effet.

Ainsi, concernant le sujet qui nous intéresse ici, la notion d'alimentation de laboratoire reposant sur l'utilisation d'un bloc d'alimentation USB-C prend tout son sens, il ne s'agit pas simplement de 500 mA en 5 volts (ou 3 A dans le cas de l'USB-C non compatible « PD ») qu'on suppose habituellement lorsqu'on parle d'USB. Bien entendu, si j'en parle, c'est précisément parce que le matériel que nous allons explorer dans un instant est compatible USB-PD et qu'il devra être utilisé avec une source compatible, comme le matériel UGREEN que j'évoquais précédemment (pas question de brancher l'alimentation sur un port USB de PC, sauf exception, et encore).

1. PRISE EN MAIN DU DP100

L'alimentation de laboratoire DP100 d'ALIENTEK se trouvera relativement facilement sur les sites habituels (Amazon, AliExpress et consorts). Selon la source d'approvisionnement (et donc le délai de livraison), ce petit joujou vous coûtera entre 60 € et 85 €. Le prix varie également en fonction des accessoires accompagnant le matériel et j'ai personnellement opté pour la solution « lente », mais économique en achetant directement cela dans le *store* ALIENTEK sur AliExpress **[1]**. Pour tout dire, notez que j'ai eu vent de ce matériel via une vidéo **[2]** de l'incomparable Dave Jones (EEVblog) analysant le circuit et procédant à quelques tests et mesures relativement satisfaisantes de son point de vue.

Le DP100 se présente sous la forme d'un petit pavé (95 mm x 62 mm x 16 mm) disposant d'un minuscule écran couleur (24 mm x 12 mm) et d'une interface (trois boutons + molette) sur l'un des côtés, incliné de 45°. Niveau connectique, l'alimentation se fait donc en USB-C via un bloc compatible USB-PD ou via un bloc « standard », maximum 32 V, via un adaptateur USB-C vers connecteur coaxial (*barrel jack*) de 5,5 mm livré avec le produit. Juste à côté de l'USB-C, on trouve une prise femelle USB type A destinée à la connexion avec un ordinateur (mode périphérique) ou à l'alimentation USB 5 V (mode hôte 5 V/1 A). Un câble USB type A mâle/mâle (~1 m) est livré avec le DP100. Remarquez que la documentation livrée avec le produit parle également de la possibilité de connecter une souris USB sur ce port pour contrôler l'interface. C'est... original.

De l'autre côté, on retrouve les assez classiques emplacements pour fiches « banane » accueillant les câbles silicones rouge et noir (~60 cm) fournis et terminés par des pinces croco. Les connecteurs sont dévissables afin de permettre la connexion de fils dénudés par pincement, mais aussi, et surtout, sont étoffés d'un anneau transparent s'éclairant (rouge et bleu) lorsque la sortie est active.

Au niveau de l'interface utilisateur et en dehors du fait que la taille de l'écran nécessite d'avoir de très bons yeux, le tout est relativement utilisable, à défaut d'être intuitif. Le « manuel » livré avec le matériel (anglais/mandarin) résume très bien la navigation dans les menus et on retiendra en premier lieu que les trois malheureux boutons possèdent

plusieurs modes d'utilisation : simple pression, longue pression et double pression. Il est possible de régler les paramètres (courant et tension) de l'alimentation directement (bouton ■ et molette) ou d'utiliser les 10 présélections (ou profils) configurables via une pression longue sur ▸. La modification d'un profil se fait ensuite avec ■ pour passer d'un champ à l'autre, ▸ pour choisir le chiffre et la molette pour varier la valeur. Les profils permettent de configurer la tension, le courant, la limite de tension (OVP pour *Over Voltage Protection*) et celle de courant (OCP pour *Over Current Protection*).

L'accès au menu des préférences se fait par double pression sur ■, puis on naviguera avec la molette pour ajuster le paramètre désiré (■ et ▸). Les entrées du menu sont :

- ***Exit*** : quitter la configuration ;
- ***OVP Set*** : tension max (protection) ;
- ***OCP Set*** : courant max (protection) ;
- ***OPP Set*** : puissance max (protection) ;
- ***OTP Set*** : température max (protection) ;
- ***REP Set*** : protection contre l'inversion de polarité ;
- ***AUTO_out*** : sortie active automatiquement ;
- ***PD_Volt*** : tension demandée en USB-PD (entrée) ;
- ***Blk Set*** : intensité lumineuse de l'écran ;
- ***Vol Set*** : volume du bip ;
- ***USB Mode*** : choix du mode pour l'USB-A ;
- ***Language*** : langue (anglais/mandarin) ;
- ***Theme Col*** : thèmes graphiques de l'interface (clair/sombre) ;
- ***Restore*** : réinitialise à la configuration d'usine ;
- ***Ver_info*** : information de version du *firmware*.

L'activation de la sortie se fait par une simple pression sur le bouton ▸ et l'écran présente alors les valeurs de tension et de courant mesurées ainsi que le mode (« CC » pour courant constant et « CV » pour tension constante). Enfin, notez que noyé dans la documentation, nous avons la double pression sur ◂ pour marche/arrêt et la double pression sur ▸ pour basculer d'une tension USB-PD à une autre.

Vous l'aurez compris, la tension fournie en sortie dépendra totalement de celle en entrée. Le DP100 ne sortira pas de 20 V à partir d'une alimentation ne fournissant que 5 V (comme un adaptateur USB classique). En USB-C et avec USB-PD, la négociation de la tension en entrée dépendra de ce que peut fournir l'adaptateur secteur et de la configuration. Il m'est arrivé plus d'une fois de maladroitement presser plusieurs fois trop rapidement sur ▸, pour ensuite me demander pourquoi l'alimentation ne fournissait pas la tension attendue. La tension USB-PD demandée avait simplement basculé de 20 V à 9 V. Je trouve la double fonctionnalité de ce bouton (activation + choix USB-PD) peu judicieuse.

À l'usage et après un petit temps d'adaptation, l'alimentation s'avère fort utile, qu'il s'agisse de fournir du courant à un montage, surveiller la consommation ou même manuellement charger un accu. Bref, tout ce qu'on fait généralement avec un matériel plus encombrant. Bien entendu, tout n'est pas parfait, le DP100 alimenté avec un bloc USB-C/USB-PD de qualité sortira une tension et un courant stable et propre, mais la présence d'une unique sortie sera très limitante pour certains usages. La compacité du produit est à la fois un avantage, faible encombrement sur un bureau « normal » et un handicap, étant donné la faible lisibilité des informations présentées à l'écran. Mais pour ce point en particulier, il existe une solution...

2. INTERFACE AVEC UN ORDINATEUR

Comme précisé en début d'article, le DP100 peut être connecté à un PC, un Mac ou une Raspberry Pi. Non pour lui fournir une alimentation, mais pour permettre le contrôle de l'appareil. ALIENTEK met à disposition une application dédiée pour Windows sur le site officiel **[3]** et, d'après la documentation **[4]**, celle-ci permet un accès plus ergonomique aux contrôles de l'appareil avec graphiques et préréglages, ainsi que la mise à jour du *firmware*. Je n'ai pas testé cette version pour deux raisons : l'accès au site était d'une lenteur catastrophique, et je n'ai aucune intention d'utiliser une application propriétaire dont je ne suis pas sûr à 100 %, même dans une VM ou un environnement émulé (Wine). De plus, l'intérêt d'une application proposant une interface graphique est, selon moi, très limité. Un tel programme doit permettre d'automatiser des opérations ou pouvoir être utilisé pour atteindre cet objectif. Ce qui n'est absolument pas le cas avec une GUI, aussi jolie et clinquante soit-elle...

L'idée étant de pouvoir télécommander l'alimentation, comme on le ferait avec un instrument compatible SCPI (*Standard Commands for Programmable Instruments*) tel le DP832 de RIGOL, il nous faut un outil, au minimum en ligne de commande, de manière à pouvoir filtrer la sortie avec les outils classiques UNIX (`sed`, `grep`, `ask`, etc.) et donner des ordres en composant un script invoquant la commande. L'utilitaire n'a donc pas besoin d'être directement scriptable, même si ceci serait un avantage certain. Dans ce genre de situation, le réflexe par défaut est de s'orienter vers GitHub/GitLab pour s'enquérir d'un développement préexistant.

Mais avant cela, commençons tout simplement par brancher le DP100 sur une machine GNU/Linux (PC ou SBC comme une Raspberry Pi), pour voir ce que le système en pense :

```
$ dmesg
[...]
new full-speed USB device number 22 using xhci_hcd
New USB device found, idVendor=2e3c,
   idProduct=af01, bcdDevice= 2.00
New USB device strings: Mfr=1, Product=2,
   SerialNumber=3
Product: ATK-MDP100
Manufacturer: ALIENTEK
SerialNumber: 223AF571C000
hid-generic 0003:2E3C:AF01.000D: hiddev0,hidraw4:
   USB HID v1.10 Device [ALIENTEK ATK-MDP100]
   on usb-0000:04:00.4-2.1.3.4/input0
```

Le matériel, identifié par les VID/PID USB `0x2e3c:0xaf01`, est énuméré comme étant un périphérique USB HID, ce qui n'est guère étonnant puisque HID, en plus d'être le protocole pour les claviers/souris, est généralement aussi celui utilisé pour toutes sortes de « gadgets », du notificateur de mail au lance-missile USB en passant par les clés de sécurité et gestionnaires de mot de passe matériels.

Même sans savoir quel outil nous finirons par utiliser, la première chose à faire est de gérer correctement les permissions sur le périphérique. Ceci passe par la configuration d'une règle *udev* nous évitant de devoir utiliser l'identité du super-utilisateur `root` pour lire/écrire sur le périphérique (entrée `/dev/hidraw4` ici, pour l'accès HID « brut »).

Le produit est livré avec un jeu de câbles de relativement bonne facture. Notez le câble USB type A mâle/mâle permettant la connexion à un ordinateur ou SBC. Ce type de câble n'est pas courant, donc... à ne pas perdre.

Notre règle, placée par exemple dans un fichier `alientekDP100.rules` dans `/etc/udev/rules.d`, ressemblera à ceci (sur une ligne) :

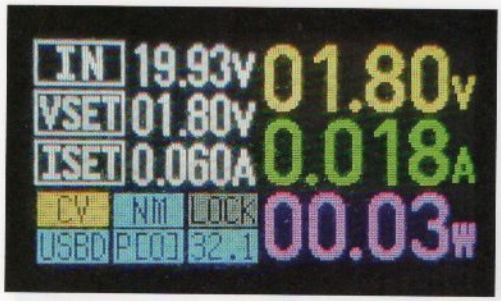
```
KERNEL=="hidraw*", ATTRS{idVendor}=="2e3c",
ATTRS{idProduct}=="af01", MODE="0660", GROUP="plugdev"
```

Ceci aura pour effet, lors de la connexion du DP100 reconnu via ses identifiants USB pour le sous-système/support `hidraw`, d'ajuster les permissions afin de permettre aux membres du groupe `plugdev` de lire et d'écrire l'entrée correspondante dans `/dev`. J'en profite d'ailleurs pour signaler que bon nombre de documentations en ligne précisent les nouvelles permissions à appliquer avec `MODE="0666"`. Ceci est totalement stupide, car revient à autoriser les opérations de lecture et d'écriture pour le propriétaire (`root`), les membres du groupe `plugdev`, et... **n'importe qui d'autre** (*others*). Oui, c'est plus simple, mais pourquoi ne pas faire directement `sudo chmod -R 666 /*` dans ce cas ?!

Quoi qu'il en soit, une fois cette nouvelle règle ajoutée, on pourra demander à *udev* de relire sa configuration et réappliquer immédiatement les règles avec :

```
$ sudo udevadm control --reload-rules
$ sudo udevadm trigger
```

Un simple `ls -l` devrait vous confirmer que le périphérique est maintenant utilisable à souhait par l'utilisateur courant (membre de `plugdev` en principe).



Zoomé ainsi, l'écran du DP100 est bien agencé et lisible, étant donné le nombre d'informations affichées. Seul petit problème, il ne fait en réalité que 24 mm de large pour 12 mm de haut.

Si nous retournons vers GitHub à présent, et cherchons « DP100 », nous trouvons quelques développements intéressants… qui ne concernent pas ce qui semble être une certification Microsoft. Parmi lesquelles :

- WebDP100 **[5]** : un début de projet en TypeScript destiné à fonctionner dans un navigateur supportant WebHID (Chrome, Chromium, Edge, etc.). Dans une vidéo liée à la page, l'auteur précise qu'il n'a aucune intention de développer une interface complète.
- Alientek-DP100-PyQT5 **[6]** qui est un *fork* traduit en anglais d'un projet identique écrit en Python avec une interface graphique Qt5 assez similaire à l'application du constructeur. Ce code repose sur l'utilisation des fichiers DLL du fabricant et possède donc une composante propriétaire (en plus d'être une application GUI).
- pydp100 **[7]**, une poignée de scripts Python assez basiques.
- « Alientek DP100 experiments » **[8]**, un tout début de module, très récent, écrit en Go qui se limite, pour l'instant, à l'envoi et la réception de rapports HID.
- open_dp100 **[9]** qui est un outil utilisable, écrit en Rust, accompagné de quelques explications (en chinois) détaillant le protocole utilisé.

C'est vers ce dernier code que nous allons nous diriger, car c'est clairement ce qui se rapproche le plus d'un utilitaire répondant à nos besoins et nous permettra éventuellement de développer davantage le concept ou, plutôt, de réimplémenter le tout dans un autre langage plus en alignement avec nos préférences, étant donné que le dépôt ne contient pas de licence et le code doit donc supposé être non réutilisable en l'état.

Installer Rust sur un système GNU/Linux, ARM ou x86, est un jeu d'enfant puisqu'il suffit de récupérer et invoquer un script shell qui fera tout le travail à votre place, tel que détaillé sur le site officiel : `curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh`. La simple exécution de cette commande vous résumera les options d'installation et vous demandera le type d'installation souhaité. Choisissez simplement « `Proceed with standard installation` », l'environnement Rust sera installé (via `rustup`) dans `~/.cargo` et une ligne ajoutée automatiquement à votre `~/.bashrc`. Ceci fait, déconnectez-vous et reconnectez-vous (ou utilisez `source "$HOME/.cargo/env"` directement) pour rendre le tout utilisable. Si vous n'avez plus besoin de Rust par la suite, vous pourrez supprimer les éléments installés en effaçant les répertoires `~/.cargo` et `~/.rustup`, ainsi que la ligne ajoutée au `.bashrc`.

Nous pouvons alors nous placer dans un répertoire quelconque et cloner le dépôt « open_dp100 » avec `git` et directement compiler et procéder à un premier essai :

```
$ git clone https://github.com/lessu/open_dp100.git
Clonage dans 'open_dp100'...
remote: Enumerating objects: 58, done.
remote: Counting objects: 100% (58/58), done.
remote: Compressing objects: 100% (40/40), done.
remote: Total 58 (delta 21), reused 47 (delta 15), pack-reused 0
Réception d'objets: 100% (58/58), 21.66 Kio | 4.33 Mio/s, fait.
Résolution des deltas: 100% (21/21), fait.

$ cd open_dp100

$ cargo build
    Updating crates.io index
  Downloaded cc v1.0.98
  Downloaded cfg-if v1.0.0
[...]
Finished `dev` profile [unoptimized + debuginfo]
   target(s) in 3.60s

$ ./target/debug/cli ls
Device count: 1
1 ATK-DP100 sn:0AA71805 hdw_ver:1.4
  app_ver:1.4 2024-04-03

$ ./target/debug/cli status
Device 0 name:ATK-DP100
Basic Info:
  vin:19.963V
  vout:0V
  iout:0A
  vo_max:19.34V
  temp1:31.5℃
  temp2:31.4℃
  dc_5v:5.067V
  out_mode:2
  work_st:0

Basic Set <9>: Off
  vo_set:15V
  io_set:5A
  ovp_set:5V
  ocp_set:5.05A
```

Comme vous pouvez le constater, nous obtenons assez facilement les informations en provenance du matériel, se résumant, plus ou moins, à ce qui est déjà affiché sur son écran. En branchant une LED (rouge 8 mm) et en réglant 4 V/20 mA avant d'activer la sortie, nous obtenons :

```
Basic Info:
[...]
  vout:1.805V
  iout:0.019A
[...]
  out_mode:0
```

Et en changeant pour 1,8 V/60 mA, nous avons :

```
Basic Info:
[...]
  vout:1.802V
  iout:0.019A
[...]
  out_mode:1
```

`out_mode` correspond donc respectivement à : `2` pour sortie désactivée, `0` pour courant constant (ici, 20 mA) et `1` pour tension constante (ici, 1,8 V). Ceci est déjà très utile dans l'état, car nous pouvons alors très simplement, dans un script shell, récupérer cette information au fil du temps pour accumuler des données et procéder à une sorte de surveillance, avec un journal et même produire un graphique si nécessaire.

L'outil cependant ne semble pas fini ou du moins pas au point d'en faire quelque chose d'autre de « publiquement » utilisable. Il n'y a, par exemple, qu'une gestion très partielle des erreurs sur les commandes passées en argument. Une commande erronée affichera un message intelligible avec une suggestion, tout comme `help` fournira une liste (partielle) de commandes, mais omettre totalement l'argument provoquera une erreur non gérée :

```
$ ./target/debug/cli
thread 'main' panicked at src/cli.rs:254:14:
internal error: entered unreachable code
note: run with `RUST_BACKTRACE=1` environment
      variable to display a backtrace
```

Côté contrôle, les opérations semblent limitées par le protocole utilisé ou, du moins, par l'état actuel du travail d'analyse de ce dernier. Nous pouvons par exemple activer/désactiver la sortie avec `set on` ou `set off` ou modifier les valeurs courantes (`set v=1.30 i=0.020 ov=10.00 oc=1.00`), ou une seule (`set v=1.05`), ou encore activer un des profils (`set config=0`). Il est également possible de combiner ces opérations (`set v=2.28 i=0.025 on`), mais les possibilités s'arrêtent là. Il ne semble ainsi pas possible de modifier les profils enregistrés ou de changer les préférences globales stockées dans l'appareil. Notez que le fichier

`100_Protocol.md`, inclus avec les sources, détaille ce qui est connu du protocole et ce qui est testé/vérifié.

Cet outil est, je pense, très intéressant, en particulier pour quelqu'un ayant des affinités avec Rust et sera donc capable de faire évoluer le code et de proposer des modifications à son auteur. Pour un usage dans un script, il fournit déjà le nécessaire, mesure et contrôle, et permettra d'atteindre notre objectif.

3. FONCTIONNEMENT EN RÉSEAU

« *Mais m'sieur, y a pas de réseau avec le DP100 !* » me direz-vous incrédule. Ce à quoi je vous répondrai « *Créature de peu de foi ! Quand on veut, on peut.* » Et effectivement, on peut, de bien des manières même. Mais celle que je vais explorer ici consistera à littéralement déporter le périphérique lui-même sur une autre machine où il sera utilisable comme s'il était local. Ce petit miracle est possible grâce au support USBIP du noyau Linux et fonctionnera avec n'importe quel périphérique USB.

Le fonctionnement est relativement simple : choisir un périphérique USB sur la machine, l'exporter pour le rendre utilisable à distance et, sur un autre poste, l'attacher via le réseau. Celui-ci apparaîtra alors comme un périphérique USB local et sera pris en charge par les pilotes adéquats (USB-HIB, ici). Vous pouvez faire de même avec une clé de stockage, une webcam, une carte Arduino, un débogueur JTAG… littéralement n'importe quoi !

Le support pour cette fonctionnalité étant intégré au niveau noyau et intervenant sur du matériel, la première chose à faire est de charger un module noyau avec :

- `sudo modprobe usbip-host` sur la machine où est branché de DP100 (le serveur) ;
- `sudo modprobe vhci-hcd` (VHCI pour *Virtual Host Controller Interface*) sur celle faisant fonctionner l'utilitaire que nous venons de compiler et tester (le client).

Il faut, ensuite, installer les outils adéquats sur les deux machines via `sudo apt-get install usbip` (Debian, Raspbian, Raspberry Pi OS, Ubuntu, etc.). La règle *udev* que nous avons précédemment vue doit être en place sur la machine « cliente », puisque le périphérique semblera effectivement connecté à cet hôte. Sur le serveur, ceci n'est pas nécessaire puisque l'accès se fait, de toute façon, avec des privilèges maximum. Ici, nous partirons du principe que le DP100 est connecté à une Raspberry Pi et l'outil fonctionne sur PC.

L'alimentation du DP100 se fait via USB-C (à droite). Le port USB type A à gauche peut être configuré pour fournir 5 V/1 A (mode hôte) ou être utilisé pour la liaison avec un ordinateur (mode périphérique).

Ainsi, côté Pi, nous commençons par lister les périphériques présents :

```
$ sudo usbip list -l
- busid 1-1.1.1 (0424:7800)
   Microchip Technology, Inc. (formerly SMSC) : unknown product (0424:7800)
- busid 1-1.1.3 (2e3c:af01)
   unknown vendor : unknown product (2e3c:af01)
```

Nous voyons l'interface Ethernet de la Pi et le DP100 connecté sur le bus 1, port 1 (hub), port 1 (du hub), périphérique 3 : `1-1.1.3`. Il nous suffit alors d'utiliser cette désignation pour exporter le périphérique en question :

```
$ sudo usbip bind -b 1-1.1.3
usbip: info: bind device on busid 1-1.1.3: complete
```

L'export est fait, le noyau sait que ce périphérique est attaché au support USBIP, mais nous devons également permettre l'accès distant en lançant le serveur gérant les connexions :

```
$ sudo usbipd
usbipd: info: starting usbipd (usbip-utils 2.0)
usbipd: info: listening on 0.0.0.0:3240
usbipd: info: listening on :::3240
```

Notez qu'il est possible d'automatiser cela en créant un service pour lancer ce démon et l'intégrer à la configuration actuelle de Systemd. Personnellement, et vu mon dédain pour ce système d'Init, je préfère m'en abstenir. Côté PC, nous pourrons alors lister les périphériques mis à disposition par le serveur :

```
$ sudo usbip list -r raspilaser.local
Exportable USB devices
======================
- raspilaser.local
    1-1.1.3: unknown vendor : unknown product (2e3c:af01)
           : /sys/devices/platform/soc/3f980000.usb/usb1/1-1/1-1.1/1-1.1.3
           : (Defined at Interface level) (00/00/00)
```

J'utilise ici le nom de la machine (résolution mDNS), mais vous pourrez tout aussi bien utiliser son IP directement. On voit clairement que le périphérique `2e3c:af01` est exporté et utilisable. Cette première connexion apparaît côté serveur dans le terminal où est lancé le démon `usbipd` :

```
usbipd: info: connection from 192.168.0.154:57356
usbipd: info: received request: 0x8005(5)
usbipd: info: exportable devices: 1
usbipd: info: request 0x8005(5): complete
```

On pourra alors attacher le périphérique en utilisant le nom ou l'IP du serveur et en spécifiant le chemin vers le matériel dans l'arborescence USB distante :

```
$ sudo usbip attach -r raspilaser.local -b 1-1.1.3
```

Encore une fois, le serveur nous informe de l'opération :

```
usbipd: info: connection from 192.168.0.154:37086
usbipd: info: received request: 0x8003(5)
usbipd: info: found requested device: 1-1.1.3
usbip: info: connect: 1-1.1.3
usbipd: info: request 0x8003(5): complete
```

À partir de ce moment, tout le reste du système côté PC réagit exactement comme si le périphérique venait d'être branché physiquement. On retrouve une trace de cette détection dans la sortie de la commande `dmesg` où il est fait mention à la fois de `vhci_hcd` se présentant comme un nouveau contrôleur USB et du périphérique USB HID comme avec une utilisation locale. La commande `lsusb` listera d'ailleurs le DP100 parmi les périphériques locaux :

```
$ lsusb
[...]
Bus 001 Device 003: ID 2109:2817 VIA Labs, Inc. USB2.0 Hub
Bus 005 Device 004: ID 2e3c:af01 ALIENTEK ATK-MDP100
Bus 005 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 004 Device 002: ID 2109:0817 VIA Labs, Inc. USB3.0 Hub
[...]
```

Et, bien sûr, notre utilitaire pour le DP100 fonctionnera exactement comme avant :

```
$ ./target/debug/cli status
Device 0 name:ATK-DP100
Basic Info:
  vin:19.963V
  vout:0V
  iout:0A
  vo_max:19.33V
  temp1:31.8℃
  temp2:31.7℃
  dc_5v:5.061V
  out_mode:2
  work_st:0

Basic Set <0>: Off
  vo_set:1.8V
  io_set:0.06A
  ovp_set:0.06V
  ocp_set:5.05A
```

Une fois les manipulations terminées, vous pourrez détacher le périphérique côté PC avec :

```
$ sudo usbip detach -p 00
usbip: info: Port 0 is now detached!
```

puis arrêter le serveur côté Pi (Ctrl+C) avant de mettre fin à l'export :

```
$ sudo usbip unbind -b 1-1.1.3
usbip: info: unbind device on busid 1-1.1.3: complete
```

Cette technique vous permettra de manipuler de manière totalement transparente un périphérique USB à travers le réseau local, mais fonctionnera exactement de la même manière via un VPN, par exemple. Non testé ici, il semblerait que ceci fonctionne également entre un hôte Linux (serveur) et un client WSL sous Windows, afin de contourner l'absence d'accès direct au matériel dans cet environnement virtualisé. Et, encore une fois, on ne parle clairement pas que du DP100. Les possibilités sont absolument incroyables...

CONCLUSION

J'avoue que j'aime beaucoup ce matériel qui se glissera dans la trousse à outils facilement et constitue une solution « de bureau » très pratique. Bien entendu, ceci ne remplacera jamais une « vraie » alimentation de laboratoire comme la RIGOL DP832 (400 €) également pilotable en USB et LAN, mais proposant une interface plus agréable et plusieurs sorties contrôlables. Le prix, quelque 60 €, est un tantinet élevé à mon sens, bien que la qualité soit au rendez-vous. Pour quelque 10 ou 15 euros de moins, la question de la pertinence de l'achat ne se poserait même pas...

Voilà pour l'aspect purement pratique. Mais la partie la plus tentatrice à mon avis est l'outil de contrôle USB HIB qui ne demande qu'à être réimplémenté en ajoutant une myriade de fonctionnalités. Je pense naturellement à un développement en C, langage que je connais et aime, mais Go, Zig ou même Python peuvent être des pistes

Ceci est un bloc d'alimentation UGREEN Nexode Travel 65 W compatible USB-PD (~30 €), avec deux ports USB-C et un USB-A. Il est capable de fournir 5 V/3 A, 9 V/3 A, 12 V/3 A, 15 V/3 A et 20 V/3,25 A. Je l'utilise à la fois pour le DP100, alimenter des montages (USB-A), charger mon smartphone Samsung A40 et alimenter mon laptop Lenovo E480.

intéressantes. On peut envisager d'intégrer un support Lua (voir article sur le sujet dans Linux Magazine 269 **[10]**) par exemple, pour rendre le tout scriptable, mais également la création de journaux de mesures, la génération de graphiques ou même le développement de greffons pour différents systèmes ou applications. Les idées ne manquent pas, le temps... oui, toujours. **DB**

RÉFÉRENCES

[1] https://fr.aliexpress.com/item/1005005992326848.html

[2] https://www.youtube.com/watch?v=Pd6LG7iP2GQ

[3] https://www.alientek.com/download

[4] https://akizukidenshi.com/goodsaffix/DP100_manual.pdf

[5] https://github.com/scottbez1/webdp100

[6] https://github.com/ketukil/Alientek-DP100-PyQT5-english-gui

[7] https://github.com/palzhj/pydp100

[8] https://github.com/jonathangjertsen/dp100

[9] https://github.com/lessu/open_dp100

[10] https://connect.ed-diamond.com/gnu-linux-magazine/glmf-269/embarquez-un-peu-de-lua-dans-vos-projets-c

UN OSCILLOSCOPE À PÉDALE

Fabien Marteau
Front de libération des FPGA

Les électroniciennes et électroniciens sont des humains comme les autres, ils ont deux mains, deux pieds et une tête. Quand il s'agit de faire des mesures avec les deux sondes de l'oscilloscope, les deux mains sont vite prises. Comment peut-on encore appuyer sur les boutons de l'engin alors que toutes nos mains sont occupées ? Et si nous utilisions nos pieds ? À l'heure de la démocratisation du vélo utilitaire, il est temps d'ajouter une pédale à votre oscilloscope.

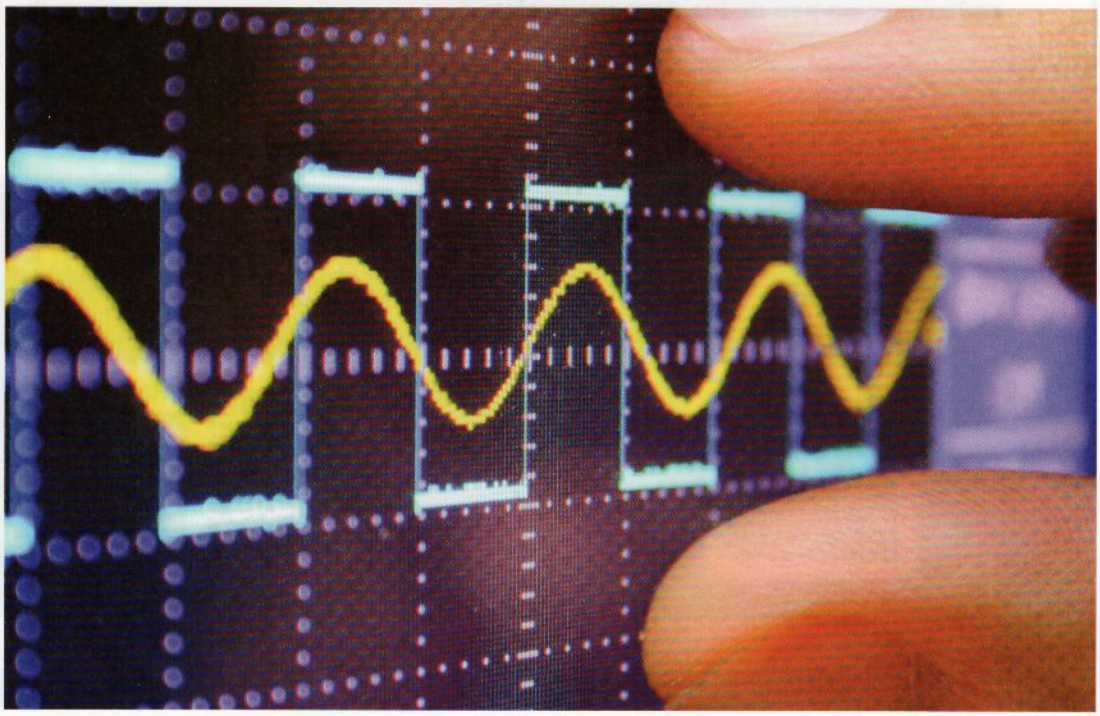


Fig. 1 : Oscilloscope ? Vélo ? Véloscope ?

Pour bien démarrer votre nouveau montage, vous avez branché le 5 V sur une alimentation stabilisée plutôt que sur l'USB, histoire de bien voir la consommation et de limiter le courant en cas de problème de court-circuit.

Vous avez également bien pris soin de régler votre oscilloscope pour qu'il déclenche sur front descendant du TX de l'UART que vous allez *monitorer* avec une première sonde dans une main et vous *monitorez* le RX avec la seconde sonde dans l'autre main. L'objectif de cette mise en route est de pouvoir saisir le chronogramme de la première trame TX envoyée par votre montage.

Tout est prêt, vos deux mains sont prises par les sondes de l'oscilloscope, la base de temps est réglée correctement pour voir passer les caractères à 115200 bauds et vous voyez bien passer la trame. Vite vite vite, il faut stopper l'oscilloscope pour qu'il ne redéclenche pas sur les trames suivantes !

Et là, c'est le drame. Si vous lâchez une sonde pour appuyer sur le bouton, l'oscilloscope va déclencher à nouveau et vous perdrez la trame voulue. Si vous restez comme ça à tenir vos sondes, vous aurez les trames suivantes, mais pas celle que vous vouliez. Vous pouvez bien essayer d'appuyer sur le bouton avec le nez ou le coude, mais l'acrobatie est loin d'être garantie. Évidemment, vous n'avez pas pu régler l'oscilloscope en mode « single » pour qu'il s'arrête

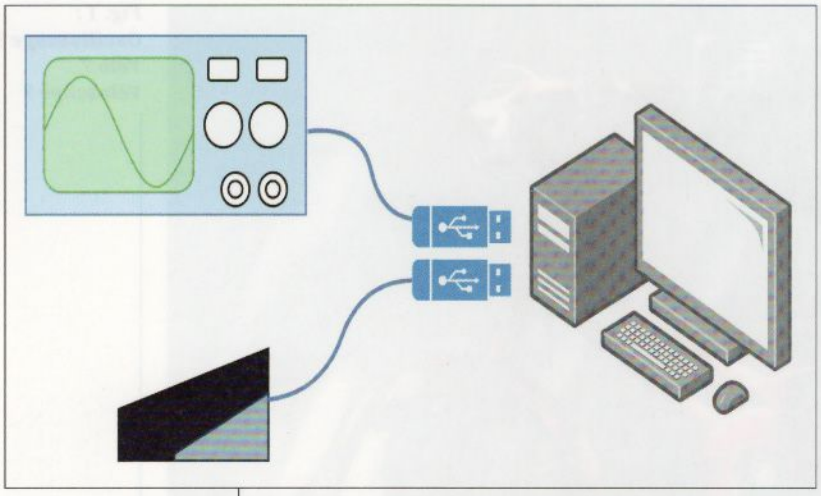


Fig. 2 : Branchement de l'oscilloscope à pédale. L'oscilloscope ainsi que la pédale sont branchés sur un PC qui se chargera de piloter le premier sur pression du pied.

au premier déclenchement, puisqu'au démarrage votre montage envoie un *glitch* qui aurait déclenché avant la trame TX de l'UART.

Alors, que faire ?

En général, le problème est résolu en demandant à Gégé de la compta de venir donner un coup de main, ou à junior, si c'est un bricolage perso dans sa cave. Mais nous n'avons pas toujours quelqu'un à disposition pour bricoler, et puis ce n'est pas un rôle ultra palpitant que de se tenir à côté de la bricoleuse ou du bricoleur pour appuyer sur le bouton sur ordre, le moment venu.

Les électroniciennes et électroniciens ne se sont pas encore reproduits suffisamment longtemps entre eux pour se transformer en Shiva avec deux bras supplémentaires dans le dos. Cependant, la plupart sont dotés de deux **pieds** en plus de leurs deux bras, pourquoi ne pas les utiliser ?

Dans cet article, nous allons voir comment ajouter une pédale USB à notre oscilloscope, de manière à déclencher le bouton STOP de l'appareil avec le pied. Nous utiliserons un oscilloscope actuel supportant le standard **VISA** et muni d'un port de contrôle sur USB que nous brancherons à un ordinateur sous GNU/Linux. Nous y ajouterons une pédale (trouvée sur la célèbre plate-forme chinoise pour une dizaine d'euros) qui se branche sur USB.

Nous allons également voir que toutes les briques logicielles permettant de faire ce montage sont déjà disponibles et documentées. Ça n'est même pas un *hack*. Il est possible d'ajouter une pédale à n'importe quel oscilloscope moderne, du moment qu'il soit connectable à un ordinateur et que le protocole soit documenté et au standard VISA.

Le pilotage d'instrument de mesure par USB a déjà été traité par Jean-Baptiste Vioix dans les colonnes de Hackable **[1]** avec des appareils **RIGOL**. Yann Guidon a également écrit un article pour piloter l'alimentation de laboratoire **AX-6002P** de marque **AXIOMET [2]**.

Nous utiliserons ici un oscilloscope **Siglent**. Comme il est également compatible avec le standard VISA, nous pourrons nous référer à l'article.

1. BRANCHEMENT DE L'OSCILLOSCOPE

L'oscilloscope utilisé pour cet article est un **SDS1202X-E** conçu par l'entreprise **Siglent**. Cet oscilloscope numérique est pilotable par USB et Ethernet.

Voyons ce que donne le branchement de l'USB sous GNU/Linux (Mint 20) :

```
[73208.563506] usb 1-1: new high-speed USB device number 4 using xhci_hcd
[73208.716657] usb 1-1: config 1 interface 0 altsetting 0 bulk endpoint 0x81
has invalid maxpacket 64
[73208.716665] usb 1-1: config 1 interface 0 altsetting 0 bulk endpoint 0x1
has invalid maxpacket 64
[73208.716670] usb 1-1: config 1 interface 0 altsetting 0 endpoint 0x82 has
an invalid bInterval 0, changing to 7
[73208.724524] usb 1-1: New USB device found, idVendor=f4ed, idProduct=ee3a,
bcdDevice=99.99
[73208.724531] usb 1-1: New USB device strings: Mfr=1, Product=2,
SerialNumber=3
[73208.724535] usb 1-1: Product: SDS1202X-E
[73208.724539] usb 1-1: Manufacturer: Siglent
[73208.724543] usb 1-1: SerialNumber: SDS1EDEQ3R4790
[73208.868510] usbcore: registered new interface driver usbtmc
```

L'appareil semble bien reconnu par le noyau qui reconnaît le nom du produit, le fabricant ainsi que son numéro de série `SDS1EDEQ3R4790`. Comme c'est un branchement par USB, nous avons également un numéro de vendeur (`idVendor`) en hexadécimal 0xf4ed ainsi qu'un numéro de produit (`idProduct`) 0xee3a.

Comme pour toute interface ouverte et standardisée, il existe un module Python associé à VISA : **PyVISA**. Nous allons donc pouvoir simplement piloter l'appareil en Python.

2. PYVISA

VISA pour *Virtual Instrument Software Architecture* est une bibliothèque standard poussée par **National Instruments**. Elle est téléchargeable sur le site **NI [3]**, mais pas vraiment libre. Il est beaucoup plus simple de se jeter sur la « version Python » *open source* **PyVISA [4]**.

PyVISA est un utilitaire Python permettant de piloter tout plein d'instruments, dont notre oscilloscope. Il est bien sûr disponible dans toute bonne distribution GNU/Linux à base de **Debian**.

```
$ sudo apt install python3-pyvisa python3-pyvisa-py python3-usb
```

La lectrice ou le lecteur qui chercherait à comprendre toutes les couches impliquées dans le pilotage des appareils de mesure en électronique pourra se référer à l'article de Denis Bodor **[5]**, pour que les sigles VISA, SCPI, GPIB, etc., n'aient plus de secret pour vous.

Avant de tenter une quelconque connexion à l'appareil, il est important de configurer ses règles `udev` correctement en ajoutant le fichier suivant :

```
$ sudo vim /etc/udev/rules.d/70-siglent.rules
```

Avec les identifiants USB de l'appareil vus au branchement. Dans le cas du **Siglent**, l'identifiant vendeur est 0xf4ed et 0xee3a pour le produit.

```
# SIGLENT SDS1202X-E
SUBSYSTEMS=="usb", ACTION=="add", ATTRS{idVendor}=="f4ed",
ATTRS{idProduct}=="ee3a", GROUP="usbtmc", MODE="0660"
```

Si le groupe `usbtmc` n'existe pas, on l'ajoute :

```
$ groupadd usbtmc
$ sudo usermod -aG usbtmc nomutilisateur
```

Pour que l'ajout du groupe à l'utilisateur soit effectif, il est important de se « délogger » de sa session pour se « relogger ».

Puis on recharge les règles avant de brancher l'USB de l'oscilloscope.

```
sudo udevadm control --reload
```

Pour les premiers tests, on peut lancer une console Python de type **ipython** :

```
$ ipython
In [1]: import pyvisa

In [2]: rm = pyvisa.ResourceManager()

In [3]: rm.list_resources()
Out[4]: ('USB0::62701::60986::SDS1EDEQ3R4790::0::INSTR',)
```

On retrouve l'oscilloscope dans la liste des ressources avec des identifiants vendeurs et produits en décimal.

Mais comme nous n'avons qu'un appareil de branché à l'ordinateur, il n'est pas nécessaire de se poser de question sur lequel utiliser. Pour ouvrir la ressource il suffira de passer `USB0::62701::60986::SDS1EDEQ3R4790::0::INSTR` en paramètre de la fonction `open_resource()`.

```
In [5]: inst = rm.open_resource('USB0::62701::60986::SDS1EDEQ3R4790::0::INSTR')
In [6]: print(inst.query("*IDN?"))
Siglent Technologies,SDS1202X-E,SDS1EDEQ3R4790,1.3.26
```

On récupère ainsi une instance `inst` sur laquelle on va effectuer des requêtes `query()` pour lire des valeurs. Pour envoyer des commandes, on utilisera la méthode `write()` avec la commande VISA à lancer.

Dans l'exemple ci-dessus, nous avons lancé une requête d'identification pour connaître le nom du constructeur, le modèle, le numéro de série ainsi que la version du *firmware* de l'oscilloscope.

Toutes les commandes VISA supportées par l'oscilloscope sont décrites dans le manuel nommé « SIGLENT Digital Oscilloscopes Remote Control Manual » et disponible gratuitement en PDF sur le site du constructeur **[6]**.

Fig. 3 : L'oscilloscope est en mode capture automatique en continu, le bouton est vert.

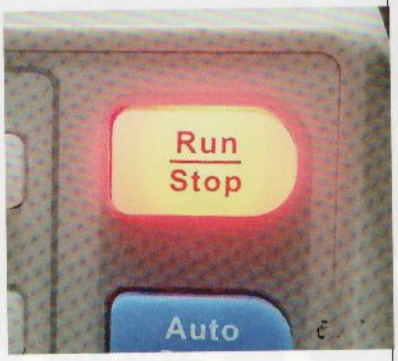


Fig. 4 : L'oscilloscope est stoppé, le bouton est rouge.

On y trouve la requête `*IDN?` qui permet de récupérer les caractéristiques d'identification de l'appareil.

On peut évidemment régler par commande Python les paramètres que l'on règle habituellement avec les boutons de l'oscilloscope. Cette manière de faire est d'ailleurs très pratique pour faire des bancs de test ou de mesure. Mais dans notre cas, nous souhaitons simplement ajouter un bouton pilotable au pied, donc nous continuerons à faire les réglages directement sur l'oscilloscope en appuyant sur le bouton **Run/Stop** pour qu'il s'affiche en vert comme on peut le voir sur la photo de la figure 3.

Une fois la capture du signal effectuée, nous souhaitons « figer » le signal en réappuyant sur ce même bouton qui fait alors office de bouton « STOP », comme en figure 4.

La documentation nous dit que si nous voulons simuler cet appui, il suffit d'écrire la commande VISA `STOP` :

```
In [7]: inst.write("STOP")
```

Et l'oscilloscope s'arrête, le bouton passe au rouge.

Nous avons compris comment piloter l'oscilloscope de manière à le faire appuyer sur le bouton `Run/Stop` et stopper l'acquisition. Passons maintenant à la deuxième partie de cette manipulation : l'interface pied-pédale.

3. PÉDALE USB

En traînant sur des sites asiatiques bien connus, on trouve toutes sortes de pédales qui se branchent sur l'ordinateur par USB pour une petite centaine de yuans.

Dans ce montage, nous utiliserons la pédale nommée *USB Foot Switch Single Pedal* proposée par la marque **PCSensor** sur AliExpress **[7]** montrée en figure 5.

Au branchement, la pédale est reconnue par le *kernel* avec le message suivant :

Fig. 5 : Une photo de la pédale trouvée sur AliExpress avec un lien QR code pour celles et ceux qui seraient intéressés.

```
[ 8261.979592] usb 1-1.3.3: new full-speed USB device number 22 using ehci-pci
[ 8262.090171] usb 1-1.3.3: New USB device found, idVendor=3553, idProduct=b001,
bcdDevice= 0.00
[ 8262.090176] usb 1-1.3.3: New USB device strings: Mfr=1, Product=2,
SerialNumber=0
[ 8262.090177] usb 1-1.3.3: Product: FootSwitch
[ 8262.090179] usb 1-1.3.3: Manufacturer: PCsensor
[ 8262.092395] input: PCsensor FootSwitch Keyboard as /devices/
pci0000:00/0000:00:1a.0/usb1/1-1/1-1.3/1-1.3.3/1-1.3.3:1.0/0003:3553:B001.0009/
input/input24
[ 8262.151777] input: PCsensor FootSwitch Mouse as /devices/
pci0000:00/0000:00:1a.0/usb1/1-1/1-1.3/1-1.3.3/1-1.3.3:1.0/0003:3553:B001.0009/
input/input25
[ 8262.151934] hid-generic 0003:3553:B001.0009: input,hidraw1: USB HID v1.11
Keyboard [PCsensor FootSwitch] on usb-0000:00:1a.0-1.3.3/input0
[ 8262.152573] input: PCsensor FootSwitch as /devices/pci0000:00/0000:00:1a.0/
usb1/1-1/1-1.3/1-1.3.3/1-1.3.3:1.1/0003:3553:B001.000A/input/input26
[ 8262.152654] hid-generic 0003:3553:B001.000A: input,hidraw2: USB HID v1.10
Device [PCsensor FootSwitch] on usb-0000:00:1a.0-1.3.3/input1
```

Linux la reconnaît directement comme un clavier et un appui sur la pédale correspond à la touche `x` d'un clavier. Nous pourrions bien sûr nous contenter de cette configuration. Cependant, la détection de l'appui d'une touche en Python nécessite l'utilisation de librairies compliquées qui ne sont pas toujours portables.

L'idée n'est pas de passer des heures à chercher la librairie idéale qui fera la capture bloquante de l'appui sur la pédale. Si on peut se contenter d'un `input()` simple de Python3, on sera content :

```
...
#Python va bloquer ici jusqu'à un appui sur x<entrée>
appuie_pedale = input("x")
inst.write("STOP")
...
```

La valeur entrée à l'appui de la pédale nous importe peu finalement. Ce qu'il nous manque par contre, c'est le caractère de fin de ligne... un appui sur la touche Entrée, quoi.

Plutôt que de correspondre à la touche `x`, il faudrait pouvoir configurer la pédale pour qu'elle appuie sur la touche Entrée, tout simplement.

Pour configurer la pédale, un CD contenant le pilote du périphérique est fourni avec le produit. Cependant :

- c'est un pilote prévu pour Windows ;
- et qui a encore un lecteur de CD-ROM accessible dans son PC « de tous les jours » ?

La flemme de sortir son lecteur CD USB du tiroir et de lire le CD-ROM pour voir un logiciel de configuration nous pousse donc naturellement à regarder comme ça vite fait sur Internet si on ne trouverait pas quelque chose.

Et en effet, on trouve un projet *open source* de configurateur de « footswitch » sur GitHub, proposé par **Radoslav Gerganov** qui fonctionne sous GNU/Linux.

```
$ git clone https://github.com/rgerganov/footswitch.git
$ cd footswitch
```

HID (HUMAN INTERFACE DEVICE)

La classe de périphérique HID est un standard défini pour l'USB. C'est le protocole qui va permettre de piloter tout ce qui est clavier, souris, manette, joystick...

Tant que le périphérique est utilisé de manière standard comme un clavier ou une souris, nous n'avons pas à nous en soucier. Mais dès que nous avons besoin de configurations spécifiques, il faut s'intéresser au protocole. C'est l'objet du programme `footswitch` de Radoslav Gerganov.

Avant de pouvoir compiler le logiciel, il faudra s'assurer d'avoir installé la version de développement de la bibliothèque HID :

```
$ sudo apt install libhidapi-dev
$ make
$ sudo make install
```

La commande `make install` ajoute le fichier de règles udev `19-footswitch.rules` dans le répertoire `/etc/udev/rules.d`. Il sera donc nécessaire de les relire avant de rebrancher la pédale :

```
$ sudo udevadm control --reload-rules && sudo udevadm trigger
```

Le dépôt `footswitch` comporte un répertoire `debian` qui permet de construire le paquet Debian et de l'installer proprement dans sa distribution (histoire d'éviter d'en mettre partout avec le `sudo make install`).

```
$ sudo apt install dpkg-dev cmake libhidapi-dev pkg-config devscripts equivs
$ cd debian
$ sudo mk-build-deps -i
$ dpkg-buildpackage -us -uc -b
```

Une fois que la pédale est installée puis branchée, on peut la lister avec l'option `-r` :

```
$ footswitch -r
[switch 1]: unconfigured
[switch 2]: x
[switch 3]: unconfigured
```

`footswitch` détecte trois commutateurs, mais seul le 2 semble configuré. Pour changer de touche, on indiquera le numéro de *switch* à configurer et l'option `-k`.

Par exemple, si l'on veut *mapper* la lettre `a`, nous utiliserons la commande suivante :

```
$ footswitch -2 -k a
```

L'appui sur la pédale correspondra désormais à la lettre `a`. Notez que cette configuration est inscrite directement dans la mémoire de la pédale. Ça n'est pas une configuration du *driver* de l'ordinateur. La pédale est reconnue comme un clavier, donc si on la branche sur un autre ordinateur, elle affichera également `a`.

Il y a cependant un problème pour les personnes qui utilisent un *mapping* de clavier exotique (en bépo, par exemple). La lettre écrite ne correspondra pas toujours à celle qui sera *mappé* lors de l'appui sur la touche.

Dans notre cas, ça n'est pas très grave, car ce qui nous importe, c'est de pouvoir appuyer sur la touche `<ENTRÉE>` dont le *mapping* n'est pas très exotique, en général.

Pour demander la touche Entrée, il suffit de la nommer comme ceci :

```
$ footswitch -2 -k enter
```

Et nous gagnons une magnifique touche Entrée actionnable avec le pied.

4. UN PETIT SCRIPT PYTHON POUR ASSEMBLER TOUT ÇA

Maintenant que nous savons piloter l'oscilloscope et que nous avons configuré la pédale pour qu'elle corresponde à l'appui sur la touche Entrée, nous pouvons faire un petit script *quick-and-dirty* pour mettre en œuvre l'oscilloscope à pédale.

```
from usb.core import USBError ❶
import pyvisa

SERIALNUM="SDS1EDEQ3R4790"
IDVENDOR = 0xf4ed
IDPRODUCT = 0xee3a
SIGLENT_OSC_NAME = f'USB0::{IDVENDOR}::{IDPRODUCT}::{SERIALNUM}::0::INSTR'
```

```
rm = pyvisa.ResourceManager()
inst = rm.open_resource(SIGLENT_OSC_NAME) ❷

footswitch_press = input() ❸

try:
    inst.query("STOP") ❹
except USBError: ❶
    pass
```

On se connecte à l'oscilloscope via le `ResourceManager` et sa méthode `open_resource()` ❷. Puis on « bloque » sur une demande de valeur entrée au clavier avec la fonction native de Python `input()` ❸. L'appui sur la touche Entrée a pour effet de passer à la ligne suivante qui est ❹ : l'envoi de la requête `STOP`.

Sur l'ordinateur de l'auteur, l'interface USB a tendance à lever fréquemment une erreur d'USB (`USBError`) alors que la requête a bien fonctionné. Il est donc nécessaire de la capturer pour l'ignorer ❶.

Pour le faire fonctionner, nous réglerons l'oscilloscope comme souhaité, puis nous lancerons le script :

```
$ python push_stop_siglent.py
```

Lorsque nous aurons la courbe voulue sur l'oscilloscope, un appui sur la pédale (donc la touche Entrée) stoppera la capture.

Bien sûr, il sera nécessaire de recommencer la manipulation à chaque fois que nous relancerons la capture sur l'oscilloscope.

Pour éviter cela et faire en sorte que le script se « réarme » automatiquement, nous pourrions le faire tourner en boucle.

```
footswitch_press = input()
while footswitch_press.strip() == '': ❶
    print("STOP") ❷
    try:
        inst.query("STOP")
    except USBError:
        pass
    footswitch_press = input() ❸
```

La modification se fait à la fin du script, on boucle sur la valeur tapée au clavier ❶. Si elle est vide (seulement la touche Entrée), on affiche tout de suite le mot `STOP`, histoire de voir une réaction à l'écran ❷. on envoie la requête d'arrêt, puis on se bloque à nouveau sur l'attente de la touche Entrée ❸.

De cette manière, il n'est plus nécessaire de revenir à l'ordinateur pour relancer un cycle après avoir relancé l'oscilloscope. De plus, on peut quitter la boucle très simplement en entrant une valeur avant d'appuyer sur la touche Entrée de son clavier.

```
$ python push_stop_loop_siglent.py

STOP

STOP
adieux monde cruel !
$
```

Tous les codes présentés ici se trouvent sur le dépôt de l'auteur **[8]**.

CONCLUSION

Malgré son aspect *quick-and-dirty*, ce montage est parfaitement fonctionnel. Nous avons pu ajouter une troisième « main » (qui s'avère être un pied) à l'oscilloscope. Grâce aux interfaces (à peu près) ouvertes, nous avons pu élaborer ce *hack* de manière assez transparente.

Nous avons utilisé une pédale USB simple pour ajouter une seule fonctionnalité. Il est cependant parfaitement possible d'utiliser un pédalier USB comportant deux ou trois pédales pour décupler les possibilités de manipulation de son oscilloscope.

Le montage nécessite cependant l'utilisation d'un PC. De nos jours, les PC peuvent être de taille réduite et avec une bonne autonomie, c'est même avec un vieux pc « ultra portable » qu'a été réalisée cette manipulation. Mais il serait intéressant de réduire la taille encore avec une **Raspberry Pi**, par exemple. La Raspberry Pi possède les entrées USB nécessaires à la connexion de l'oscilloscope et de la pédale, et comme elle tourne avec des distributions GNU/Linux récentes, il n'y a pas de problème de portage.

La Raspberry Pi est tout de même un ordinateur assez puissant qui se rapproche d'un ordinateur de bureautique. Ça reste cher pour l'ajout d'une simple touche à son oscilloscope. Surtout si on le dédit à cette fonctionnalité.

Pour réduire encore les coûts, on pourra aller voir du côté des SBC (*Single Board Computer*) chinois comme le **Milk-Duo** présenté par Denis Bodor dans le Hackable 54 **[9]**. Le temps d'intégration des *drivers* et la mise en place d'un programme seront sans doute un peu plus longs que sur un PC avec Python, mais nous restons sur un système GNU/Linux avec de l'USB. Si la partie USB complexifie le développement, on peut même passer par l'Ethernet de l'oscilloscope et une pédale sans USB (simple commutateur).

L'ultime simplification (côté utilisateur) consisterait à brancher la pédale USB directement sur l'oscilloscope, puisque l'appareil utilisé dans cet article possède une entrée USB « host » pour y brancher des clefs USB de stockage pour faire des saisies d'écran. Peut-être serait-il possible d'y brancher un autre périphérique USB ?

Cette option nécessite par contre de « hacker » (dans le sens noble du terme) l'oscilloscope de manière à adapter le comportement de la prise USB. Et ça tombe bien, quelqu'un l'a fait avec le projet **360nosc0pe [10]**. L'électronique qui constitue l'oscilloscope de Siglent est en fait **Zynq** de chez **AMD/Xilinx**. Le Zynq est un SoC (*System On Chip*) muni d'une partie processeur (ARM Cortex-A9) et d'une partie FPGA (Artix 7). Le projet 360nosc0pe utilise le langage de description matériel nommé Migen/Litex basé sur du Python et promu par l'entreprise française **Enjoy Digital**. Comme tout le code du projet est fourni, il doit être possible de l'adapter pour piloter une pédale USB, et le piloter ainsi en direct !

Attention, en utilisant ainsi une pédale pour faire de l'électronique quotidiennement, vous risquez d'y prendre goût. Mais sachez que pour satisfaire votre dépendance, il est possible de décupler son usage en vous déplaçant à vélo et ainsi venir grossir les rangs des « vélotaffeuses » et « vélotaffeurs » qui commutent quotidiennement et font leur sport réglementaire pour se maintenir en forme. Et si les rues et routes de votre région sont encore dangereuses, n'hésitez pas à rejoindre l'une des nombreuses associations de défense du vélo comme le **CADRes de Colmar et environs [11]**, dans le cas de la ville de la rédaction du magazine. **FM**

RÉFÉRENCES

[1] *Automatisez vos mesures en utilisant l'USB*, Vioix, Jean-Baptiste, Hackable 24 (mai 2018), **https://connect.ed-diamond.com/Hackable/hk-024/automatisez-vos-mesures-en-utilisant-l-usb**

[2] *Reverse-engineering d'une alimentation numérique et contrôle avec bash*, Guidon, Yann, Hackable 32 (janvier 2020), **https://connect.ed-diamond.com/Hackable/hk-032/reverse-engineering-d-une-alimentation-numerique-et-controle-avec-bash**

[3] National Instruments, **https://www.ni.com/fr/support/downloads/drivers/download.ni-visa.html**

[4] PyVISA, **https://pypi.org/project/PyVISA/**

[5] *Contrôlez votre alimentation de laboratoire avec votre Raspberry Pi*, Bodor, Denis, Hackable 15 (novembre 2016), **https://connect.ed-diamond.com/Hackable/hk-015/controlez-votre-alimentation-de-laboratoire-avec-votre-raspberry-pi**

[6] Siglent Remote Control Manual, **https://int.siglent.com/upload_file/user/SDS1000X+/SIGLENT_Digital_Oscilloscopes_Remote_Control_Manual.pdf**

[7] Foot Switch FS221-P, PCSensor, **https://fr.aliexpress.com/item/1005006279064716.html**

[8] Dépôt du code de l'article, **https://github.com/Martoni/OscilloPedale.git**

[9] *Milk-V Duo : un minuscule SBC RISC-V à 8 €*, Bodor, Denis, Hackable 54 (mai 2024), **https://connect.ed-diamond.com/hackable/hk-054/milk-v-duo-un-minuscule-sbc-risc-v-a-8-eu**

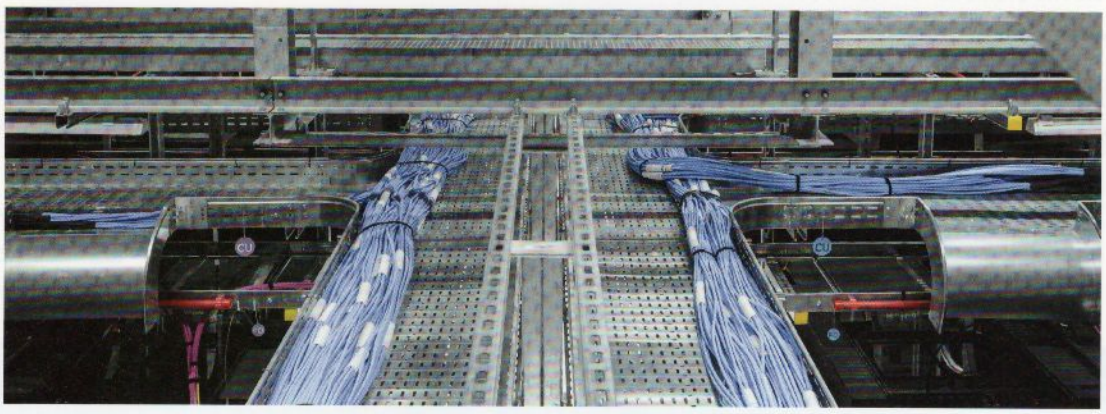
[10] 360nosc0pe, **https://github.com/360nosc0pe/scope**

[11] Cyclistes Associés pour le Droit de Rouler en sécurité Colmar et environs, **http://www.cadrescolmar.org**

CONCEVOIR, METTRE EN PLACE ET BIDOUILLER UN ENVIRONNEMENT BASÉ SUR LE PROTOCOLE INDUSTRIEL MODBUS

Erwan Cordier

Dans cet article, nous allons étudier et mettre en place un environnement de système industriel. Nous utiliserons des outils open source et le protocole de référence Modbus TCP. D'abord, nous allons observer le fonctionnement du protocole, ensuite nous mettrons en place un environnement de test pour comprendre comment détourner le comportement normal d'un microcontrôleur pour provoquer des malfonctions. Bien que largement couvert par d'autres articles des éditions Diamond et malgré son âge avancé (1re version en 1979), le protocole Modbus TCP est toujours d'actualité et reste utilisé sous plusieurs formes pour gérer des microcontrôleurs industriels (PLC) actuellement sur le marché.



– Concevoir, mettre en place et bidouiller un environnement basé sur le protocole industriel Modbus –

Nous allons proposer ici la mise en place d'un environnent qui exploite pleinement Modbus TCP pour que vous puissiez l'implémenter et expérimenter directement. Veuillez noter que l'objectif de cet article est de permettre une compréhension et l'exploitation du protocole et de son environnement, mais nous ne couvrirons pas la partie automatisation et développement des automates (Ladder Logic, Graphset, etc.).

1. INTRODUCTION

1.1 Historique et présentation du protocole Modbus

Pour une introduction plus complète du protocole, je me permets de citer celle proposée dans les lignes du *GNU/Linux Magazine n°208* concernant le protocole Modbus **[1]**.

Créé en 1979 par Modicon, Modbus est un protocole de communication industriel, utilisé initialement pour communiquer avec des automates programmables, il se décline sous deux versions :

- Modbus RTU ou Modbus ASCII pour les lignes séries ;
- Modbus TCP pour l'Ethernet.

Il fonctionne sur la logique de maître-esclave. Les esclaves sont totalement passifs et seul le maître peut initier un échange de données, selon une logique reposant sur le principe de la question/réponse. Ce protocole est principalement utilisé dans les réseaux d'automates programmables. Ce protocole facile à utiliser, implémenter et robuste est devenu un protocole industriel de référence, surtout depuis son encapsulation dans les trames Ethernet.

Dans cet article, toutes les références au protocole Modbus feront référence au protocole Modbus TCP, Modbus RTU n'est pas abordé ici.

1.2 Recherche et structure protocolaire

Le protocole Modbus, par son ancienneté, a été largement documenté et modifié pour satisfaire des demandes sur mesure, par exemple, le protocole UMAS de Schneider Electric se sert de Modbus comme base pour son propre protocole.

1.2.1 Structure protocolaire

La structure protocolaire de Modbus est volontairement simpliste.

Les exemples sont tirés de l'IETF `draft-dude-modbus-applproto-00.txt` pour une compréhension complète du protocole et de chaque code de fonction. Nous vous recommandons cette lecture.

Un paquet Modbus TCP se structure de la manière suivante :

```
|---------------------- ADU ----------------------------|
|  Header  |  Function Code  |  Data        | Error Check |
           |----------- PDU ---------------|
```

Le protocole est basé sur une pile TCP/IP. La section Modbus se trouve dans la partie TCP et est envoyée sur le port 502 (qui est le port dédié pour Modbus).

1.2.1.1 MODBUS ADU : EN-TÊTE

L'en-tête a une longueur de 7 octets et comprend les champs suivants :

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Inv_id                 |          Proto_id             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         length                |Unit Identifer |  MODBUS PDU   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- L'Inv ID (2 octets) est utilisé pour définir une paire (une forme de signature pour associer une requête et une réponse) ;
- L'ID du protocole (2 octets) est toujours 0 pour les services Modbus (un ID qui n'est pas zéro peut être lié à une extension du protocole) ;
- La taille (2 octets) correspond à la somme du champ UID additionné à la taille du PDU ;
- L'ID unitaire (1 octet) ou UID est utilisé pour identifier un serveur distant situé sur un réseau non TCP.

1.2.1.2 MODBUS PDU : TYPE DE DONNÉES, CODE DE FONCTION

Modbus base son modèle de données sur une série de fichiers de registres qui sont présents sur le système qui reçoit le paquet.

Il y a deux catégories de modèle de données :

- les *coils* (valeur booléenne 1 bit) ;
- les registres (valeur hexadécimale 16 bits) ;

Les *coils*, par exemple, peuvent être utilisés pour définir ou retourner l'état d'un interrupteur, quand les registres peuvent définir ou retourner l'état d'une unité de mesure (L , M/s, Pa). Nous parlons ici de « définir » et de « retourner » puisqu'il y a un concept d'écriture et de lecture dans la construction d'une trame Modbus qui doit être pris en compte.

Les systèmes *Input/Output* (I/O = valeur physique) n'acceptent que la lecture, quand les données d'applications acceptent la lecture/écriture.

La suite est assez logique. Nous pouvons faire deux types d'actions, lire ou écrire sur un ou plusieurs *coils* ou registres (en comptant une restriction en lecture seule sur les I/O).

Je vous affiche ici 4 possibilités de codes fonctions.

- Écriture ou lecture :
  - → *Discrete Output (Coils)* || *data from application*
  - → *Holding Registers* || *data from application*
- Lecture seule :
  - → *Discrete Input (Coil)* || *data from I/O system*
  - → *Input Register (Register)* || *data from I/O system*

Voici donc les principales actions que nous pouvons faire, chaque action est liée à son code de fonction.

Public function codes [edit]

Function type			Function name	Function code
Data Access	Bit access	Physical Discrete Inputs	Read Discrete Inputs	2
		Internal Bits or Physical Coils	Read Coils	1
			Write Single Coil	5
			Write Multiple Coils	15
	16-bit access	Physical Input Registers	Read Input Registers	4
		Internal Registers or Physical Output Registers	Read Multiple Holding Registers	3
			Write Single Holding Register	6
			Write Multiple Holding Registers	16
			Read/Write Multiple Registers	23

Pour les codes de fonctions demandant la lecture ou l'écriture de plusieurs éléments, la partie Modbus PDU est définie via une adresse de base (*start adresse*) suivie d'une valeur *n* (*quantity*) qui va être additionnée à une valeur de base. Donc, si vous voulez lire deux registres qui ne sont pas successifs, il faut faire plusieurs requêtes, là où deux registres successifs peuvent être requêtés en un seul.

1.2.2 Recherche documentaire

Dans la suite de cet article, nous allons nous poser en attaquants ayant une volonté de destruction matérielle.

La compréhension du protocole est très importante, cependant si nous voulons uniquement modifier l'état d'un bouton, il nous faut juste le code de fonction qui permet de modifier un *coil*. La même logique peut être appliquée à tous les autres protocoles : si je veux récupérer le contenu d'une page HTML, je vais utiliser le paramètre GET. La difficulté est de comprendre un paquet qui a une fonction qui n'est pas présente dans une quelconque documentation officielle.

Néanmoins, il existe quelques chemins de traverse pour trouver votre bonheur (sauf si vous êtes le premier ou un des seuls à travailler sur ce protocole). Dans les deux exemples que nous allons traiter, nous allons nous servir de l'outil Wireshark et de son dissecteur de trames intégré.

1.2.2.1 DISSECTEUR WIRESHARK INTÉGRÉ

Wireshark est un outil très connu disponible sur GitHub et sur GitLab maintenu par la Wireshark Fondation et est sous licence GNU GPL, il est donc *open source*. Un des concepts de base de Wireshark est la dissection de paquets qu'il reçoit sur une interface définie. Mais, pour qu'il puisse identifier quel protocole est employé et comment séparer les différents éléments d'un paquet, il faut qu'il puisse le définir.

Wireshark possède déjà une base de données de protocoles permettant de lire des paquets facilement, c'est là la fonction la plus intéressante de l'outil. Vous pouvez voir Wireshark comme une grosse documentation de protocoles. L'outil utilise ses bibliothèques protocolaires pour faire des comparaisons avec le contenu des paquets reçus.

Ici, nous avons un exemple de **tous les codes de fonctions listés par Wireshark** :

```
static const value_string function_code_vals[] = {
    { READ_COILS,              "Read Coils" },
    { READ_DISCRETE_INPUTS,    "Read Discrete Inputs" },
    { READ_HOLDING_REGS,       "Read Holding Registers" },
    { READ_INPUT_REGS,         "Read Input Registers" },
    { WRITE_SINGLE_COIL,       "Write Single Coil" },
    { WRITE_SINGLE_REG,        "Write Single Register" },
    { READ_EXCEPT_STAT,        "Read Exception Status" },
    { DIAGNOSTICS,             "Diagnostics" },
    { GET_COMM_EVENT_CTRS,     "Get Comm. Event Counters" },
    { GET_COMM_EVENT_LOG,      "Get Comm. Event Log" },
    { WRITE_MULT_COILS,        "Write Multiple Coils" },
    { WRITE_MULT_REGS,         "Write Multiple Registers" },
```

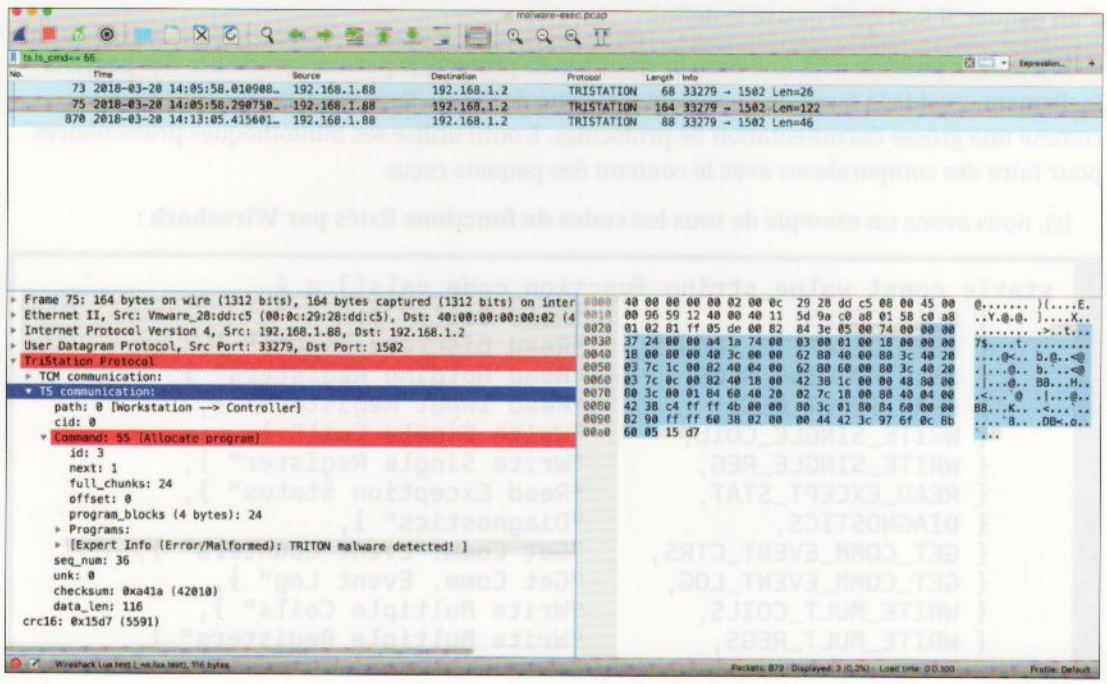
```
    { REPORT_SLAVE_ID,          "Report Slave ID" },
    { READ_FILE_RECORD,         "Read File Record" },
    { WRITE_FILE_RECORD,        "Write File Record" },
    { MASK_WRITE_REG,           "Mask Write Register" },
    { READ_WRITE_REG,           "Read Write Register" },
    { READ_FIFO_QUEUE,          "Read FIFO Queue" },
    { ENCAP_INTERFACE_TRANSP,   "Encapsulated Interface Transport" },
    { UNITY_SCHNEIDER,          "Unity (Schneider)" },
    { 0,                        NULL }
};
```

Ce qui est intéressant ici est la centralisation de l'information. Un élément supplémentaire que je trouve pertinent est le fait qu'un code de fonction nommé `UNITY_SCHNEIDER` est présent, alors qu'il n'est ni défini dans la page Wikipédia ni dans l'IETF listée plus haut. On peut donc tomber sur des éléments non documentés, mais approuvés par au moins deux personnes (le créateur du dissecteur et la fondation Wireshark).

1.2.2.2 DISSECTEUR WIRESHARK COMMUNAUTAIRE

Dans certains cas (un à ma connaissance), il arrive qu'un protocole ne soit pas documenté de manière publique et que Wireshark n'intègre pas dans sa version de base le dissecteur officiel pour ce dernier. Dans ces cas-là, vous pouvez (en général) compter sur la communauté Wireshark.

En effet, en plus de son moteur en C, Wireshark intègre un moteur de *scripting* de dissection formulable en Lua (langage de *scripting*), fait pour le développement de dissecteur sur mesure. Ce système est assez facile à prendre en main et fonctionne en dépilant au fur et à mesure le



paquet. Les cas d'usage sont assez variés et ne se résument pas uniquement à la définition de protocoles réseau, il peut aussi avoir des cas d'usage de décodages différents sur certains champs de données, ou de la détection de certains comportements malicieux.

Grâce à ce moteur, n'importe qui peut ajouter son propre dissecteur. Et en général, la communauté distribue ces dissecteurs librement sur GitHub et sur GitLab.

Par exemple, il y a quelques années l'entreprise Nozomi Networks avait développé un **dissecteur Wireshark** pour détecter le *malware* TRITON, qui se spécialisait dans l'attaque des systèmes de contrôle industriels.

2. ENVIRONNEMENT, SEGMENTATION RÉSEAU ET LOGICIELLE

Maintenant les bases du protocole Modbus comprises, attardons-nous sur l'environnement dans lequel va évoluer ce protocole. L'objectif est de simuler un environnement de technologie opérationnel.

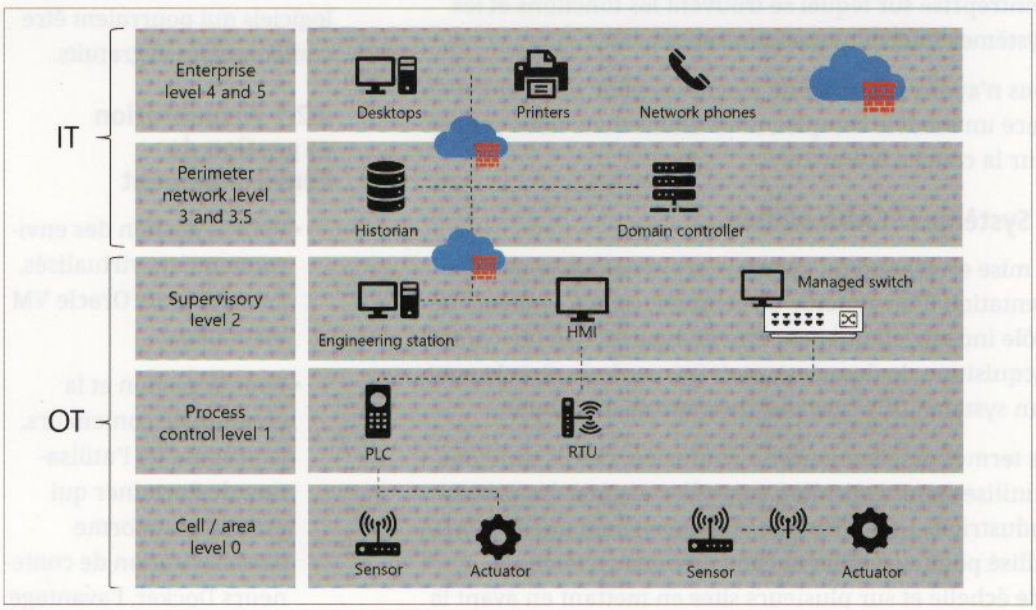
Les exigences suivantes sont obligatoires au bon fonctionnement d'un environnement OT :

- exigences en matière de délais et de performances ;
- exigence de disponibilité.

Les environnements ne sont pas les mêmes entre l'IT et l'OT. C'est pour cela que la segmentation est une préparation d'une architecture réseau nécessaire, cela nous amène donc au prochain point.

2.1 Modèle de Purdue

Le modèle de référence **Purdue** est un modèle de segmentation des réseaux de systèmes de contrôle industriels (ICS)/OT qui définit six couches, composants et contrôles de sécurité pertinents pour ces réseaux.



Exemple d'écran de représentations du modèle de Purdue.

- Niveau 0, cellule et zone :

  Le niveau 0 est constitué d'un large éventail de capteurs, d'actionneurs et de dispositifs impliqués dans le processus de fabrication de base.

- Niveau 1, contrôle du processus :

  Le niveau 1 comprend des contrôleurs intégrés qui contrôlent et manipulent le processus de fabrication et dont la fonction principale est de communiquer avec les dispositifs de niveau 0.

- Niveau 2, supervision :

  Le niveau 2 représente les systèmes et fonctions associés à la supervision et à l'exploitation du *runtime* d'une zone d'une installation de production (interface homme-machine).

- Niveaux 3 et 3,5, niveau de site et réseau de périmètre industriel :

  Le niveau de site représente le niveau le plus élevé de systèmes d'automatisation et de contrôle industriels. Les systèmes et les applications qui existent à ce niveau gèrent les fonctions d'automatisation et de contrôle industriels à l'échelle du site.

- Niveaux 4 et 5, réseaux de l'entreprise :

  Les niveaux 4 et 5 représentent le réseau du site ou de l'entreprise sur lequel se trouvent les fonctions et les systèmes informatiques centralisés.

Nous n'avons pas assez de machines pour pouvoir mettre en place un modèle complet, nous allons donc nous concentrer sur la couche 0, 1 et 2.

2.1.1 Systèmes SCADA ou DCS

La mise en place d'un système industriel passe par l'implémentation d'un système de contrôle, un ICS (système de contrôle industriel) qui peut être de type SCADA (de contrôle et d'acquisition de données), mais pas seulement, cela peut être un système DCS (système de contrôle distribué).

Ces termes peuvent porter à confusion, car ils sont tous deux utilisés pour contrôler, surveiller et gérer les processus industriels. Les différences majeures sont que le SCADA est utilisé pour surveiller et contrôler des systèmes à grande échelle et sur plusieurs sites en mettant en avant la surveillance et le contrôle en temps réel de processus en temps réel. DCS est lui utilisé pour contrôler de multiples processus et sous-systèmes individuels à partir d'un seul endroit, il a pour objectif d'avoir un contrôle plus précis grâce à sa nature localisée.

Ici, dans un objectif de simplicité d'implémentation, nous allons appliquer un système SCADA malgré la petite taille de l'environnement avec le logiciel ScadaBR.

2.2 Sélection des logiciels

Pour simplifier au maximum la réalisation de l'environnement, j'ai choisi de sélectionner uniquement des logiciels qui pourraient être conteneurisés et gratuits.

2.2.1 Virtualisation et gestion de l'environnement

- Pour la gestion des environnements virtualisés, j'ai opté pour Oracle VM VirtualBox.

- Pour la gestion et la création de conteneurs, je propose ici l'utilisation de Portainer qui est une plateforme d'orchestration de conteneurs Docker, l'avantage

ici est de pouvoir utiliser l'interface graphique, qui est plus explicite et ergonomique que l'environnement en ligne de commande de Docker.

2.2.2 Programmation, émulation et orchestration de l'automate

2.2.2.1 PROGRAMMATION

Comme expliqué plus haut, l'objectif n'est pas de développer des automates, cependant je vais quand même citer OpenPLC Editor qui est un logiciel de développement de procédés de contrôle, même si nous ne l'utiliserons pas directement.

2.2.2.2 ÉMULATION

Pour l'émulation, nous utiliserons l'autre logiciel d'OpenPLC, OpenPLC Runtime, qui servira d'émulateur de PLC.

2.2.2.3 ORCHESTRATION

Pour l'orchestration, nous allons utiliser le système de gestions ScadaBR, qui est un logiciel SCADA, dans le sens où il va pouvoir interroger le PLC, puis interpréter et afficher les données requêtées.

2.3 Architecture du lab et segmentation

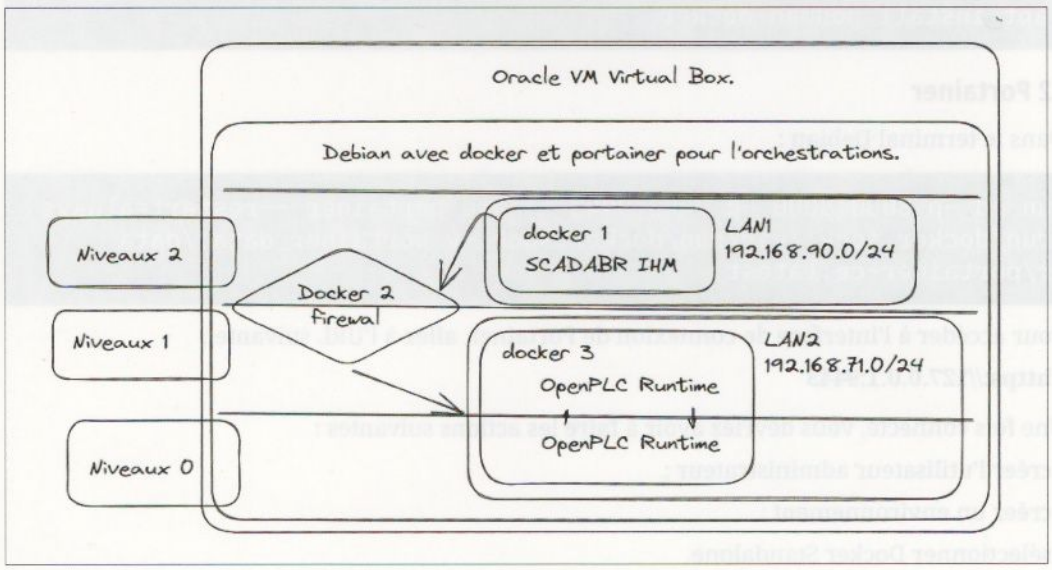
Pour l'architecture, je propose donc d'utiliser un environnement « dockerisé » (Figure ci-dessous).

3. MISE EN PLACE DE L'ENVIRONNEMENT

3.1 Installation et configuration des réseaux

Pour que Docker puisse attribuer les interfaces aux différents dockers, il faut d'abord les créer. Avec VirtualBox, il est possible de créer des interfaces locales. Voici le chemin pour y accéder :

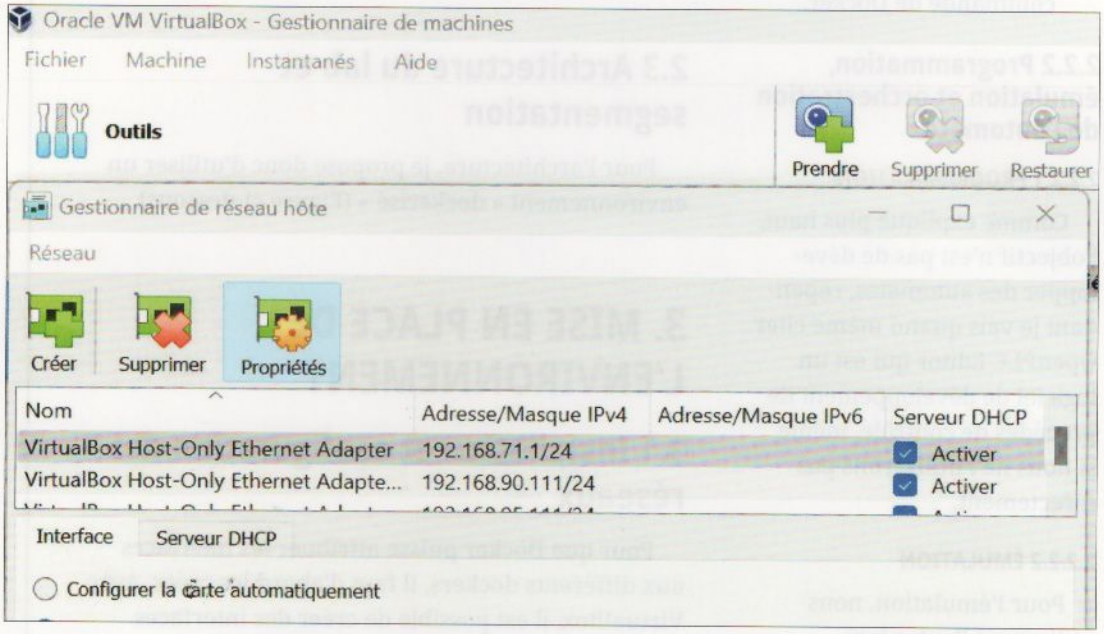
En haut à gauche du gestionnaire des machines de VirtualBox : ***Fichier*** => ***Gestionnaire des réseaux hôte***.



Nous allons créer deux interfaces de réseau hôte :

- 192.168.71.0/24 (environnement de production, niveaux 0 et 1) ;
- 192.168.90.0/24 (environnement de *monitoring*, niveau 2).

Une fois cela fait, ajouter les interfaces réseau aux paramètres réseau de votre machine.



3.1.1 Docker

```
apt update
apt install docker.io
apt install podman-docker
```

3.1.2 Portainer

Dans le terminal Debian :

```
docker run -d -p 8000:8000 -p 9443:9443 --name portainer --restart=always
-v /var/run/docker.sock:/var/run/docker.sock -v portainer_data:/data
portainer/portainer-ce:latest
```

Pour accéder à l'interface de connexion de Portainer, allez à l'URL suivante :

- **https://127.0.0.1:9443**

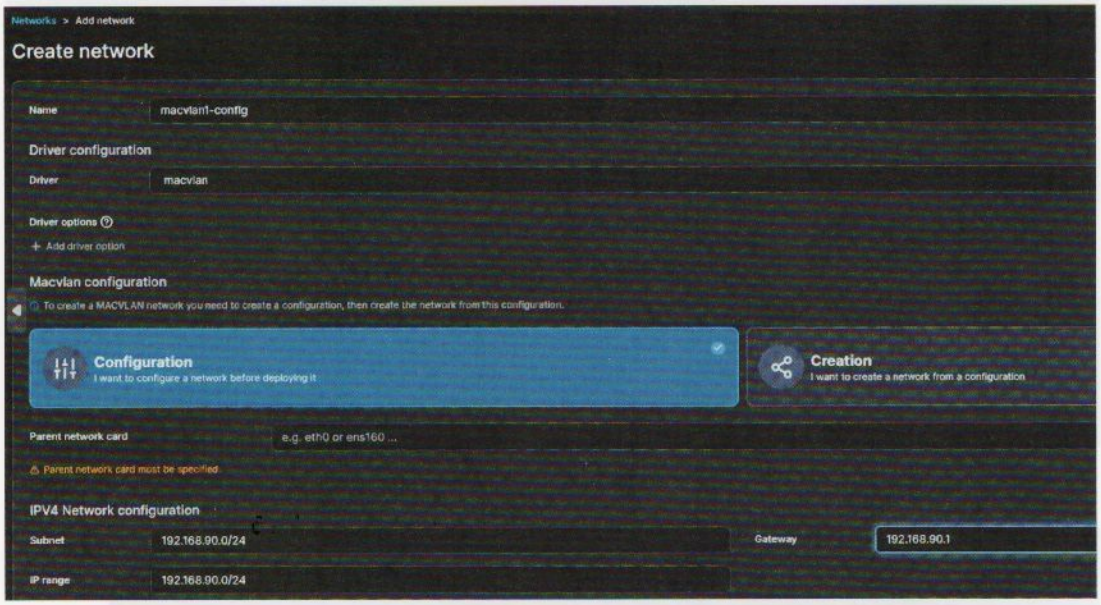
Une fois connecté, vous devriez avoir à faire les actions suivantes :

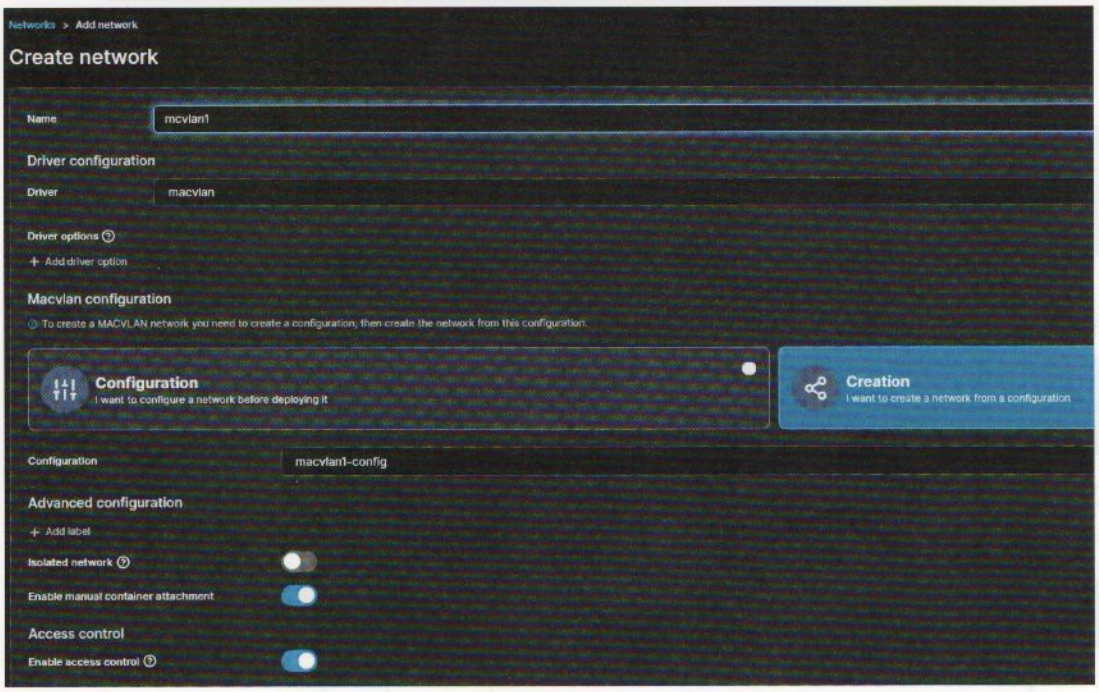
- créer l'utilisateur administrateur ;
- créer un environnement ;
- sélectionner Docker Standalone.

Nous devons assigner des adresses et segmenter le réseau entre les deux VLAN, pour cela nous allons utiliser l'option Macvlan de Docker. En résumé, Macvlan se lie à une interface réseau et assigne à Docker les adresses MAC liées à l'interface.

L'ajout de Macvlan se fait en deux étapes.

Pour commencer, vous devez ajouter un réseau de configuration (ici, l'exemple est pour 192.168.90.0/24, il faut le faire une deuxième fois pour 192.168.71.0/24).





Nous avons donc maintenant deux interfaces réseau que nous pouvons appliquer au docker :

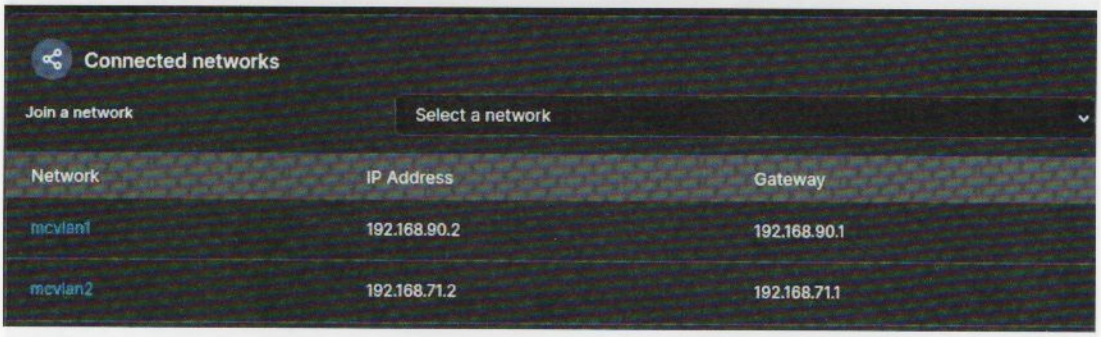
- macvlan1 (environnement de production, niveaux 0 et 1) ;
- macvlan2 (environnement de *monitoring*, niveau 2).

3.1.3 Routeur

Crée le Docker.

```
docker run --privileged -itd  vimagick/iptables
```

Attribuez les deux interfaces Macvlan au routeur.



Allez dans le TTY interactif du Docker.

Sélectionnez `/bin/sh`, et non `/bin/bash` comme sélection de TTY.

Configuration du routeur.

```
iptables -A FORWARD -i vlan1_name -o vlan2_name -j ACCEPT
iptables -A FORWARD -i vlan2_name -o vlan1_name -j ACCEPT
echo 1 > /proc/sys/net/ipv4/ip_forward
```

Si vous le souhaitez, vous pouvez changer l'adresse IP de chaque interface pour récupérer directement la passerelle par défaut.

```
ifconfig eth1 192.168.71.1 netmask 255.255.255.0
ifconfig eth0 192.168.90.1 netmask 255.255.255.0
```

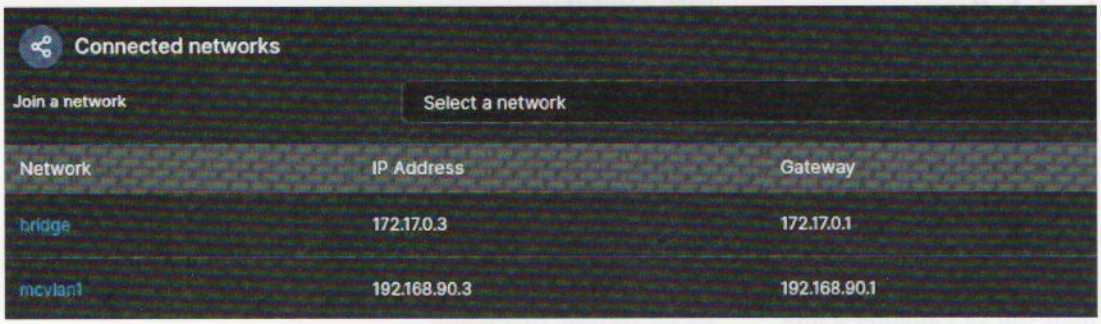
3.1.4 ScadaBR

Dans le terminal Debian :

```
docker run --privileged -p 8081:8080 -itd bitelxux/scadabr
```

Le port 8080 est modifié en 8081, car le 8080 est déjà utilisé par OpenPLC.

Configuration réseau (en bas des paramètres du *container* ScadaBR) :



Allez dans l'interface TTY.

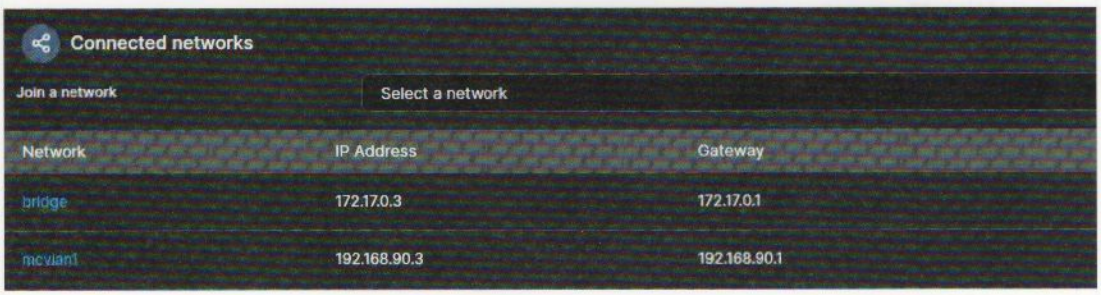
```
apt update
apt-get install iputils-ping
apt install net-tools
route add default gw IP_DU_ROUTER NOM_INTERFACE-RESEAUX
```

3.1.5 OpenPLC_V3

```
docker run -p 8080:8080 -p 502:502 tuttas/openplc_v3
```

Une fois ces dockers installés, vous pouvez les voir dans l'environnement local dans la section ***containers*** de votre interface Portainer.

Configuration réseau (en bas des paramètres du *container* OpenPLC) :



Allez dans l'interface TTY.

```
apt update
apt-get install iputils-ping
apt install net-tools
route add default gw IP_DU_ROUTER NOM_INTERFCACE-RESEAUX
```

Pour tester l'intégrité du routeur et des connexions, vous pouvez tenter de *pinger* votre ScadaBR à partir de votre OpenPLC. Si la connexion ne se fait pas, tentez de voir si les interfaces sont bien attribuées ou si les IPTables se sont bien appliquées sur le routeur.

3.2 Configurations logicielles

3.2.1 OpenPLC

Télécharger et charger une scène.

Une scène est définie ici comme un comportement préprogrammé d'un automate qui va s'adapter à une situation, p. ex. l'action d'un bras robotisé qui va soulever et baisser une caisse quand un poids défini est atteint.

Comme convenu, nous n'allons pas développer de scène, mais directement en récupérer une via l'URL de la référence **[5]**. Nous nous servirons du fichier `feux_de_signalisation.st`.

OpenPLC Editor traduit tout en code ST. C'est le langage de base d'OpenPLC.

Si vous prenez l'exemple de *Hello World* et que vous allez dans ***Fichier*** -> ***Générer un programme***, vous obtiendrez *Hello World* en langage ST.

Nous allons utiliser une scène d'exemple représentant un feu de signalisation. Pour accéder à l'interface de gestion d'OpenPLC (**http://0.0.0.0:8080/**), l'identifiant est « openplc » pour l'utilisateur et le mot de passe.

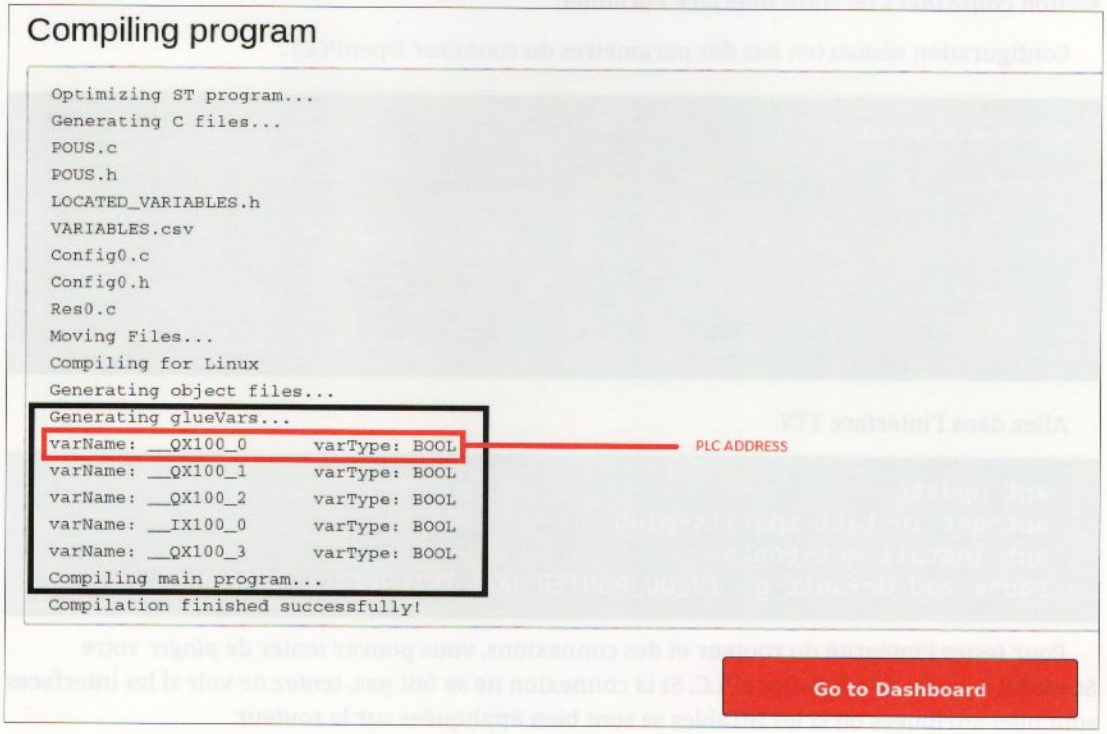
3.2.1.1 AJOUTER UNE SCÈNE

Pour ajouter une scène, suivre le chemin suivant une fois connecté à l'interface OpenPLC :

- ***Program*** -> ***Browse*** -> ***Select file*** -> choisir un nom -> ***Upload program***

Pendant la compilation du programme, nous avons des informations essentielles qui sont liées à la partie programmation de l'automate.

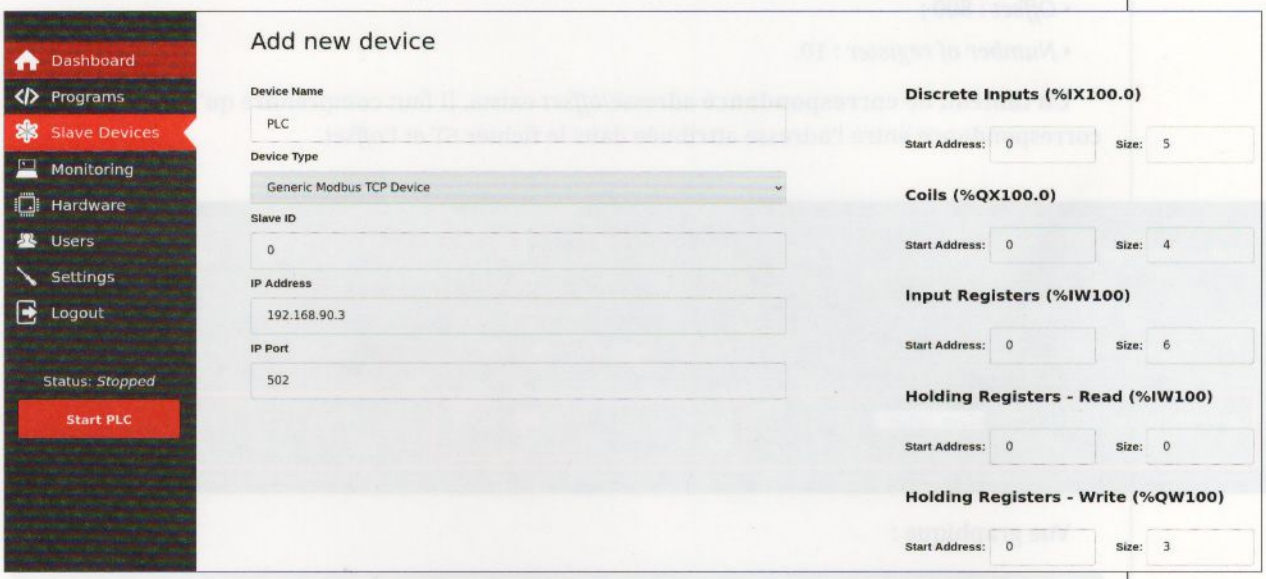
Les adresses des *coils* vont nous permettre de tester les connectivités entre ScadaBR et OpenPLC dans la partie suivante.



3.2.1.2 AJOUTER UN NOUVEAU PLC

Pour ajouter un PLC, suivre le chemin suivant une fois connecté à l'interface OpenPLC.

• ***Slave Device*** -> ***Add new device***



Pour terminer, cliquez sur le bouton ***Start PLC***.

Voilà, nous avons un PLC qui fonctionne, pour vérifier, allez dans la session ***Monitoring***, vous devriez voir les différents paramètres et leurs adresses associées représentés par des boutons qui passent du rouge au vert.

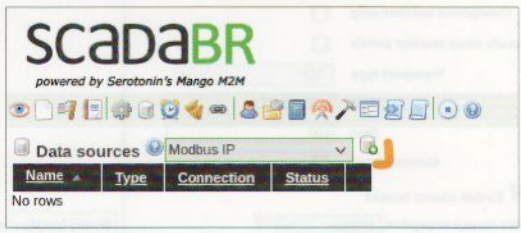
3.3 ScadaBR

Les identifiants de l'interface de connexion de ScadaBR (**http://0.0.0.0:8081/ScadaBR/login.htm**) sont :

• *user* : *admin*

• *pass* : *admin*

Pour ajouter une source de données, il faut aller à l'adresse suivante : **http://0.0.0.0:8081/ScadaBR/data_sources.shtm**. Sélectionnez ***Modbus IP*** et cliquez sur le logo de base de données avec un petit plus vert :



Ne fermez surtout pas la page qui s'affiche, on va s'en servir pour les tests de connexion.

3.4 Test de connexion

Pour confirmer la capacité de connecter l'interface SCADA au PLC, nous allons configurer ScadaBR.

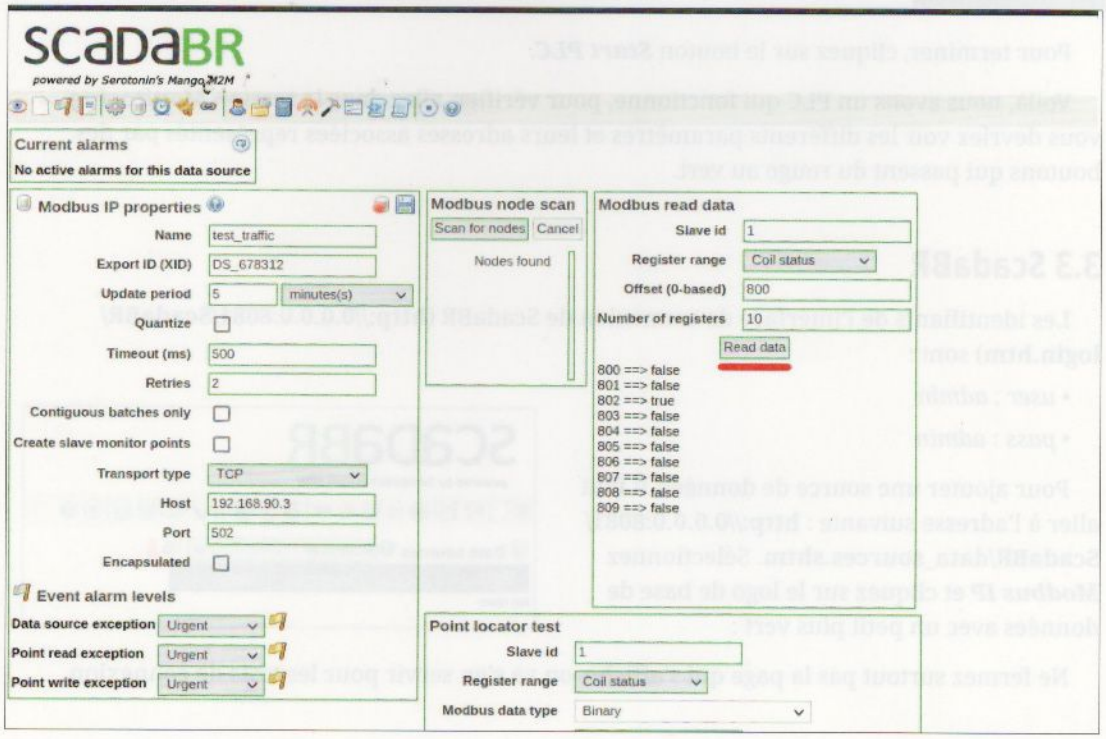
Il faut attribuer les paramètres suivants :

- IP du PLC (*host ID*) : 192.168.90.3 ;
- *Slave ID* : dans l'exemple est à 1, si cela ne marche pas, testez avec 0 ;
- *Offset* : 800 ;
- *Number of register* : 10.

Un tableau de correspondance adresse/*offset* existe. Il faut comprendre qu'il y a une correspondance entre l'adresse attribuée dans le fichier ST et l'*offset*.

Modbus Data Type	Usage	PLC Address	Modbus Data Address	Data Size	Range	Access
Discrete Output Coils	Digital Outputs	%QX0.0 - %QX99.7	0 - 799	1 bit	0 or 1	RW
Discrete Output Coils	Slave Outputs	%QX100.0 - %QX199.7	800 - 1599	1 bit	0 or 1	RW

Vue graphique :



À chaque fois que nous allons requêter le PLC, une requête Modbus va être forgée, demandant de requêter une série de *coils*, depuis l'adresse 800 jusqu'à l'adresse 810, vers l'IP 192.168.90.3.

4. PRÉPARATIONS ET ATTAQUE

4.1 Écoute de l'environnement et analyse des paquets

Partons du principe que pour cette étape, vous n'avez pas d'informations sur le système que vous avez déployé.

Pour avoir une vision correcte, il faut se placer sur le routeur, et commencer à écouter ce qu'il se passe sur le réseau. Cela tombe bien, nous avons installé plus tôt `tcpdump` pour pouvoir générer un fichier PCAP.

```
# cliquez plusieurs fois sur le bouton "Read data" de ScadaBR sur une
période de une à deux minutes pour avoir un bon échantillon.
tcpdump -i INTERFACE_NAME -w out_modbus.pcap
```

Pour extraire le fichier PCAP, il y a la commande `docker cp`.

```
# list docker container ID
docker ps
docker cp container_id:/foo.txt foo.txt
```

Note : vous pouvez analyser le fichier via `tshark`.

```
┌──(root㉿c220f24d63ac)-[~/spoof/OT-NAF]
└─# tshark -r out_modbus7.pcap -Y modbus
Running as user "root" and group "root". This could be dangerous.
   23  19.211847 192.168.90.3 → 192.168.71.3 Modbus/TCP 77 Response: Trans:     0; Unit:   1, Func:   1: Read Coils
   35  24.309351 192.168.90.3 → 192.168.71.3 Modbus/TCP 77 Response: Trans:     0; Unit:   1, Func:   1: Read Coils
   47  26.010916 192.168.90.3 → 192.168.71.3 Modbus/TCP 77 Response: Trans:     0; Unit:   1, Func:   1: Read Coils
   57  27.025428 192.168.90.3 → 192.168.71.3 Modbus/TCP 77 Response: Trans:     0; Unit:   1, Func:   1: Read Coils
   67  28.347164 192.168.90.3 → 192.168.71.3 Modbus/TCP 77 Response: Trans:     0; Unit:   1, Func:   1: Read Coils
   77  30.274528 192.168.90.3 → 192.168.71.3 Modbus/TCP 77 Response: Trans:     0; Unit:   1, Func:   1: Read Coils
   87  31.600950 192.168.90.3 → 192.168.71.3 Modbus/TCP 77 Response: Trans:     0; Unit:   1, Func:   1: Read Coils
   96  33.249535 192.168.90.3 → 192.168.71.3 Modbus/TCP 77 Response: Trans:     0; Unit:   1, Func:   1: Read Coils
```

4.1.1 Analyse du PCAP

Voici une capture exemple des communications entre ScadaBR et OpenPLC avec les différents éléments (voir Figure page suivante).

Avec les concepts sur le protocole développés en partie 1, nous comprenons donc que les éléments qui définissent les comportements sont définis par les codes de fonctions.

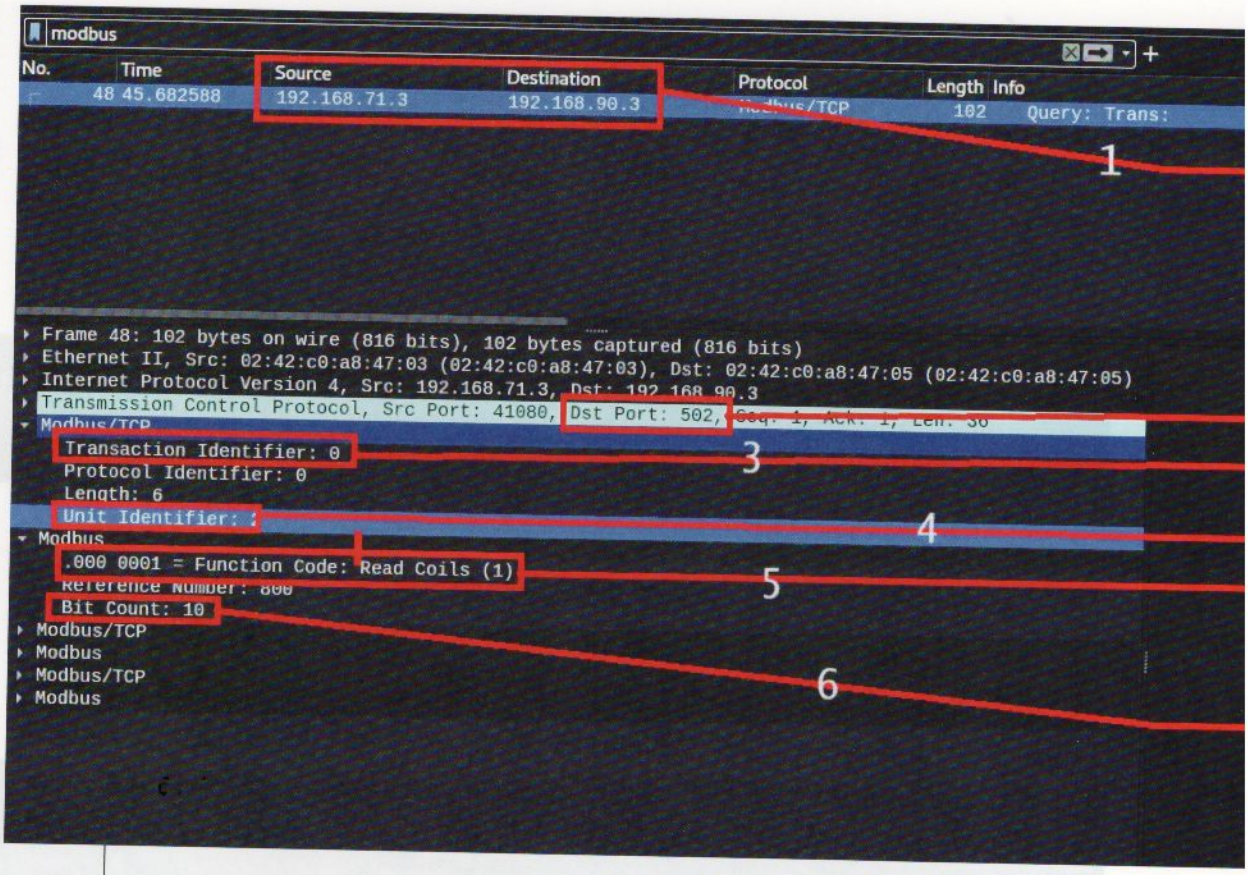
Dans Wireshark, on peut faire des recherches par code de fonction Modbus :

- `modbus.func_code == 1`

Une liste des filtres existe sur le **site de Wireshark**.

Après analyse, nous savons donc les choses suivantes sur l'environnement :

- il y a des paquets Modbus qui transitent du réseau x.x.71.x/24 au x.x.90.x/24 pour les requêtes et l'inverse pour la réponse ;



- les paquets « requête » contiennent des codes de fonctions (FC:1) qui intègrent une requête sur le statut des *coils* (par paquet de 11), à partir de l'adresse 800 (800-810) ;
- on constate qu'il y a au moins les 5 premiers *coils* qui sont actifs (valeur booléenne changeante).

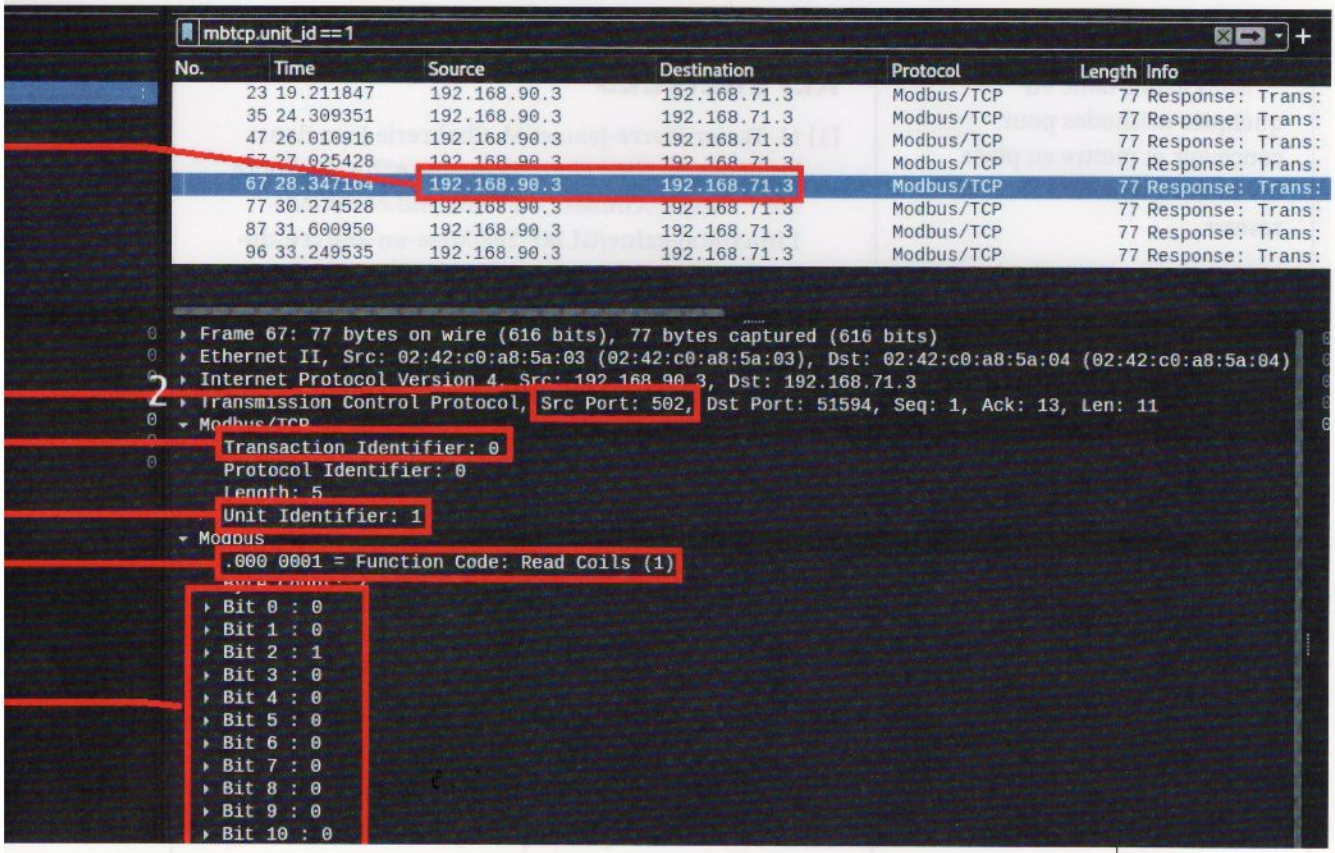
4.2 Attaque ciblée

Une fois que nous connaissons un peu plus en profondeur le fonctionnement et les registres utilisés, nous pouvons commencer à attaquer.

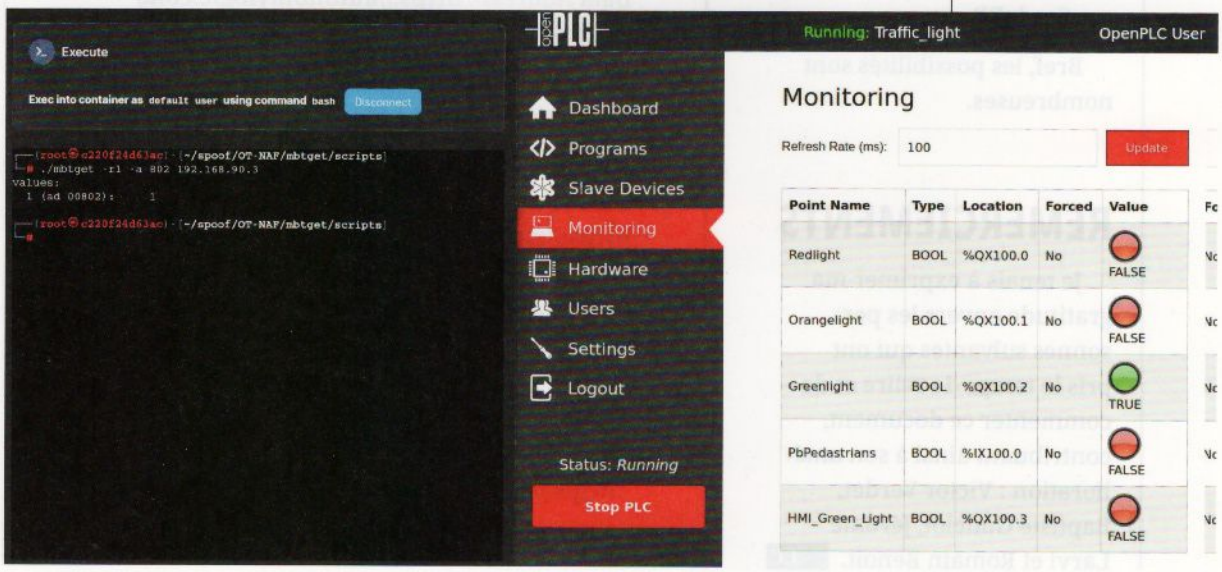
Il y a plusieurs moyens de forger des requêtes Modbus, voici une petite liste :

- **mbtget** : forge des requêtes Modbus, écrit en Perl.
- **pymodbus** : une lib Python qui permet de faire des requêtes Modbus.
- **scapy** : une lib Python qui permet de forger des *packers*.
- **gopacket** : un module Golang qui permet comme Scapy de forger des paquets.

Ici, pour la simplicité, nous allons utiliser `mbtget`.



Maintenant, je vous laisse utiliser l'option `w5` de `mbtget` pour créer un peu de chaos sur la route.



CONCLUSION

Nous avons donc vu quelques méthodes pour concevoir et mettre en place un environnement d'analyse réseau.

Mais il reste encore quelques chemins que vous pourriez aborder, comme avoir une vision plus défensive en ajoutant des règles IPtables pour fortifier le réseau ou mettre une sonde pour pouvoir faire de la détection.

Ou une approche plus offensive, comme tenter des attaques réseau de *man in the middle* pour pouvoir récupérer les paquets à la volée.

Ou encore comprendre plus en profondeur comment les automates sont développés, en créant votre propre fichier `.st` avec OpenPLC ou améliorer la représentation des données sur ScadaBR.

Bref, les possibilités sont nombreuses.

REMERCIEMENTS

Je tenais à exprimer ma gratitude envers les personnes suivantes qui ont pris le temps de relire et de commenter ce document, contribuant ainsi à son amélioration : Victor Verdet, Baptiste Gudelot, Jérôme Larvi et Romain Benoit. EC

RÉFÉRENCES

[1] M. Texier Pierre-Jean et M. Chabrerie Jean dans les lignes du *GNU Linux Magazine n°208* d'octobre 2017 : **https://connect.ed-diamond.com/GNU-Linux-Magazine/GLMF-208/Mise-en-aeuvre-du-protocole-Modbus-RTU-sur-WaRP7-via-Qt5**

[2] Base de code Wireshark pour la dissection du paquet Modbus : **https://github.com/boundary/wireshark/blob/master/epan/dissectors/packet-mbtcp.c**

[3] Dépôt GitHub du dissecteur Wireshark développé par Nozomi Networks : **https://github.com/NozomiNetworks/tricotools**

[4] Exemple de modèle de Purdue : **https://learn.microsoft.com/fr-fr/azure/defender-for-iot/organizations/media/how-to-set-up-your-network/purdue-model.png#lightbox**

[5] TinyURL pour l'accès au *drive* comportant le `.st` : **https://shorturl.at/xyKQ6**

[6] La scène d'exemple, inspirée par le travail de « seafox c » : **https://www.youtube.com/@seafoxc**

[7] Tableaux de correspondance PLC Address / Modbus Data Address : **https://autonomylogic.com/docs/2-5-modbus-addressing/**

[8] Liste des filtres Modbus pour le moteur de recherche de Wireshark : **https://www.wireshark.org/docs/dfref/m/modbus.html**

[9] URL de mbtget : **https://github.com/sourceperl/mbtget**

[10] URL de la documentation de PyModbus : **https://pymodbus.readthedocs.io/en/latest/**

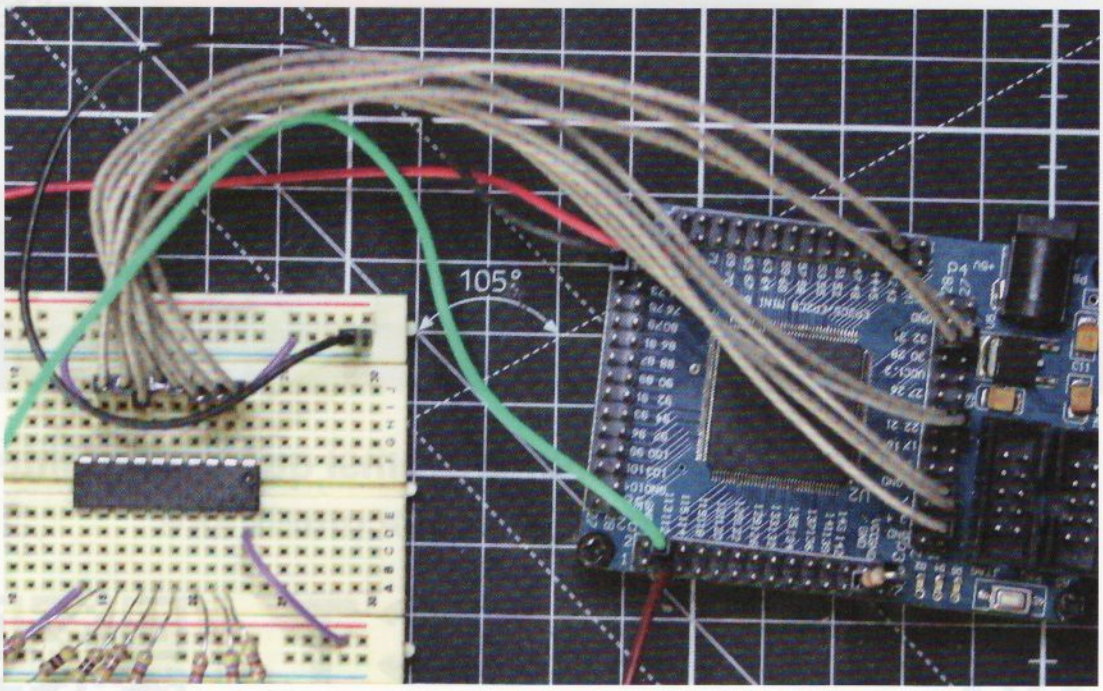
[11] URL de la documentation de Scapy : **https://scapy.readthedocs.io/en/latest/**

[12] URL de la documentation de GoPacket : **https://pkg.go.dev/github.com/google/gopacket**

MON PREMIER PROJET FPGA : UN ORDINATEUR 8 BITS COMPLET EN VHDL

Denis Bodor

Zilog a annoncé dernièrement la fin de la production du Z80 après près de 50 ans de bons et loyaux services. Nous sommes loin d'une pénurie de ces vénérables processeurs 8 bits, mais voilà l'excuse parfaite pour découvrir comment créer une architecture complète à base de Z80 dans un circuit logique programmable, et plus exactement un FPGA. L'objectif est simple : réunir tous les éléments de notre ordinateur 8 bits sur platine à essais que nous avions détaillé dans les pages des numéros précédents et lui faire exécuter un code en C !



– Mon premier projet FPGA : un ordinateur 8 bits complet en VHDL –

Les circuits logiques programmables, ou PLD (pour *Programmable Logical Devices*), représentent tout un univers (et je pèse mes mots), très différent de celui qu'on connaît par l'utilisation de microcontrôleurs ou même de processeurs couplés à de la ROM/RAM (comme le précédent projet Z80). Si vous êtes comme moi, peut-être vous êtes vous déjà essayé à l'exercice de développer un petit projet simpliste, en Verilog ou VHDL, afin de prendre en main les concepts et les subtilités du domaine, pour finalement avoir bien du mal à aller plus loin. En effet, concevoir le « *hello world* » du FPGA, à savoir une ou plusieurs LED qui clignotent grâce à un compteur, du paramétrage de l'environnement à la programmation du composant, en passant par la synthèse, le *mapping* et le routage, est relativement aisé et surtout largement documenté, quel que soit le composant choisi. Le problème, c'est « l'après », et plus exactement comment arriver à passer de ce résultat, certes très satisfaisant la première fois, à quelque chose de plus consistant et vraiment utile. C'est généralement là que, devant la nébulosité des documentations et tutoriels, noyé sous une masse de nouveaux paradigmes et termes occultes, on finit souvent par simplement jeter l'éponge. Du moins dans mon cas, et ce à plusieurs reprises, je dois l'avouer...

Ce que je vous propose ici, c'est de franchir cette marche décisive de la courbe d'apprentissage en approchant le problème d'une manière différente, naïve et potentiellement brouillonne (que les experts me pardonnent). L'idée consiste à suivre une logique identique à celle, généralement inavouée, qu'on applique souvent en programmation, lorsqu'on appréhende un nouveau langage ou une nouvelle bibliothèque : récupérer des bouts à gauche et à droite pour obtenir quelque chose de fonctionnel, pour ensuite affiner et, sur la base de quelque chose qui fonctionne et que l'on aura fait « soi-même », commencer à comprendre les tenants et les aboutissants de l'ensemble, tout en butant sur des problèmes.

1. FPGA, matériel et environnement : du propriétaire presque partout

Les lecteurs assidus du magazine savent sans doute qu'il existe des solutions entièrement *open source* permettant de simuler et de synthétiser du *bitstream* (la configuration binaire dictant au FPGA le circuit qu'il doit implémenter) à destination de certains FPGA. Fabien Marteau **[1]** nous a en effet, à plusieurs reprises, présenté du matériel et des outils qui font cela très bien, que ce soit avec l'un ou l'autre langage courant du domaine que sont Verilog et VHDL, mais également d'autres, comme Scala/Chisel **[2]**.

L'idée est cependant ici de faire avec ce qu'on a, que ce soit logiciel ou matériel. Or, ce que j'ai moi sous la main, ce sont de vieilles cartes de développement Altera d'origine « mystérieuse » (probablement eBay il y a fort fort longtemps, lors d'une précédente tentative pour m'attaquer sérieusement au sujet). Plus précisément, entre autres, une petite carte avec un Cyclone II / EP2C5T144C8 (j'ai dit que c'était vieux), un oscillateur 50 MHz, trois LED et des régulateurs de tension. Côté logiciel, nous avons un dépôt GitHub **[3]** de Joshua Bassett (alias nullobject) regroupant une triplée d'exemples pour la carte DE0-Nano de Terasic (avec un Cyclone IV / EP4CE22F17C6), incluant un *softcore* Z80, de la ROM (4 Kio) et de la RAM (4 Kio).

Je dispose d'une DE0-Nano, mais celle-ci est bien trop riche pour un tel projet (32 Mio de SDRAM, ADC, EEPROM i2c, etc.). D'autant que j'ai déjà une autre idée de réalisation en tête, impliquant une mise en œuvre plus définitive et je ne veux donc pas monopoliser une DE0-Nano à quelque 130 euros. C'est là également un autre élément important de cette petite aventure : le budget doit rester raisonnable, même pour, et en particulier pour, un achat récent.

Je vous avouerai que je n'ai pas souvenir du coût de la carte Cyclone II, ni même de celui de sa grande sœur en ma possession, à base de Cyclone IV (EP4CE6E22C8) se distinguant par la présence complémentaire d'une SDRAM H57V2562GTR de 32 Mio (provenance tout aussi mystérieuse). Ce que l'on peut constater, en revanche, c'est que des cartes/modules Cyclone IV (EP4CE15F23C8) se trouvent sur AliExpress pour une cinquantaine d'euros (avec SDRAM, LED, boutons, et même un convertisseur USB/série CH340). C'est plus qu'il n'en faut pour accueillir le projet de nullobject, même complété de quelques éléments que nous allons voir. Des cartes Cyclone II, très similaires à la mienne, sont également disponibles pour ~20 €, à condition qu'on cherche bien (utilisez le terme « EP2C5T144 » et non « Cyclone » ou « Altera »).

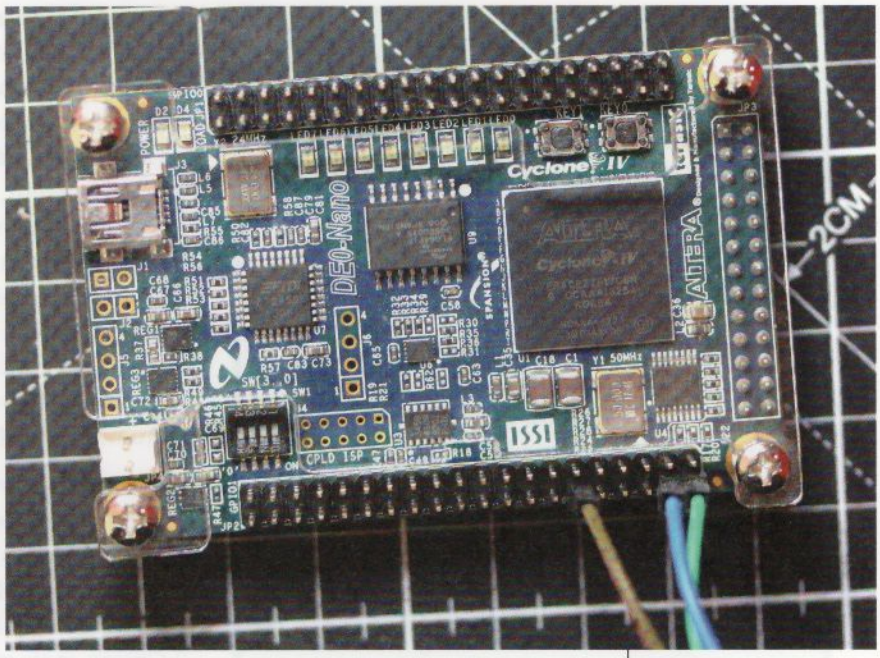
Une autre option possible, impliquant de changer de constructeur (et d'environnement), mais concernant un matériel plus moderne, est celle des cartes Tang Nano 9K de Sipeed qu'on trouvera sans peine sur les sites habituels pour un peu plus d'une quinzaine d'euros. Celle-ci pourra être utilisée avec l'environnement (propriétaire, mais gratuit pour Windows comme pour GNU/Linux en version « éducation ») du constructeur du FPGA intégré (GOWIN). Notez que la Tang Nano 4K, plus modeste, avait servi de base à Fabien pour son projet de sortie HDMI pour GameBoy dans le numéro 44 **[4]**, mais même si celle-ci se trouve encore facilement, elle est vendue au même prix que la 9K, sans doute car elle intègre également un cœur ARM Cortex M3 (ou « Coetex » selon le wiki Sipeed). Une version plus étoffée existe, c'est la Tang Nano 20k, pour quelque 40 €. J'ai commandé ce matériel, pris en main l'IDE (via son interface en ligne de commande) et porté le projet, mais n'ai pas encore reçu le matériel pour valider définitivement le tout. Nous nous en tiendrons donc ici aux composants Altera (appartenant à Intel depuis 2015, sachant qu'AMD a également acquis un fondeur de FPGA en 2022, Xilinx).

Je viens d'utiliser les mots « étoffé », « riche » et « modeste » pour désigner les FPGA. Ces composants se caractérisent par le nombre d'éléments logiques qu'ils intègrent et qui permettent de composer des circuits. Certains constructeurs utilisent le terme « LUT », d'autres « Logic Cell » et d'autres « Logic Element » (ou « LE »). Ils sont similaires mais non identiques, car cela dépend de la technologie utilisée et donc du fabricant, sans compter le marketing pour jouer avec les chiffres. Plus un FPGA possède de LUT, LE ou LC, plus il est « capable ». Voyez cela comme de l'espace pour vos circuits logiques, mais ce n'est pas tout. En plus de cette donnée, ces composants embarquent également d'autres éléments comme des PLL pour générer des fréquences sur la base de celle de l'oscillateur principal, de la mémoire, des multiplicateurs matériels, etc. Et à cela s'ajoutent le nombre d'entrées/sorties disponibles/utilisables, les autres composants présents sur la carte de développement (LED, interrupteurs, boutons, écran LCD, afficheur 7 segments, flash, ADC, capteur), etc.

En plus du matériel de base, constitué de la carte FPGA quelle qu'elle soit,

nous aurons besoin d'un convertisseur USB/série 3,3 V et d'éventuellement quelques LED (s'il n'y en a pas assez sur la carte). Mais plus important encore, nous devrons disposer d'un programmeur pour le FPGA (et/ou la mémoire flash qui accompagne le FPGA pour contenir sa configuration en cas de coupure d'alimentation). Certaines cartes ou *devkits*, comme c'est le cas pour le DE0-Nano, intègrent ceci de base, d'autres non et mettent alors à disposition un simple connecteur JTAG. Dans ce cas, il faut utiliser un programmeur externe, comme un USB Blaster pour FPGA Altera. D'autres solutions, génériques, existent, mais il vaut mieux rester cohérent, car en cas de problème, lorsqu'on découvre, avoir à douter du matériel est la dernière chose qu'on souhaite. Notez que certains vendeurs proposent une déclinaison de leur produit accompagné d'un programmeur pour quelques euros de plus (des clones économiques d'USB Blaster).

Enfin, concernant le choix de la plateforme et du composant, je tiens à souligner que cela a une importance très relative pour ce type d'apprentissage. En effet, durant la préparation de l'article et lorsque j'étais encore passablement vierge dans le domaine (en dehors du *blink*), je suis finalement arrivé à transposer mes premiers efforts du DE0-Nano au Cyclone II, puis au Cyclone IV, sans difficulté. Et également d'un FPGA Altera à un GOWIN, peu de temps après, sans trop d'encombres (sous réserve d'une validation matérielle à venir). Ceci alors même que la mémoire intégrée au FPGA (blocs M9K chez Altera et BSRAM chez GOWIN) est gérée différemment entre ces deux composants. Ce qu'on apprend avec un FPGA est à 90 % acquis pour l'ensemble des composants.



Le devkit DE0-Nano de Terasic est un classique pour l'apprentissage et la découverte du monde de FPGA, car il intègre énormément de fonctionnalités en plus du Cyclone IV lui-même. Son coût, quelque 130 €, reste cependant un investissement conséquent...

Avec le choix du composant vient celui de l'environnement qui sera, par défaut, celui du constructeur, généralement propriétaire, mais utilisable gratuitement pour un usage « éducatif » (ou ludique). Pour pouvoir télécharger, il vous faudra classiquement créer un compte sur le site (c'est le cas chez Intel/Altera), accepter les conditions, jurer sur la tête de votre chat, fournir un mot des parents, etc. Dans la très grande majorité des cas, les systèmes d'exploitation supportés sont Windows et GNU/Linux (pas de macOS, et X86/x86_64 uniquement). Attendez-vous à un volume de données conséquent, même si vous pouvez avoir une bonne

CHOIX DU LANGAGE

Il existe des langages de plus haut niveau, comme Chisel, permettant le développement pour les circuits logiques programmables, et même des solutions permettant de faire cela en Python si vous aimez (cocotb, Migen, etc.). Cependant, il me paraît plus raisonnable de tout d'abord apprendre les classiques avant de passer à quelque chose de plus « moderne ». C'est un avis tout personnel, mais tout comme connaître le C et l'assembleur avant d'attaquer Python, Rust, Zig ou Go, ceci fait davantage sens pour moi que de faire l'inverse.

Reste donc Verilog et VHDL et, là encore, c'est clairement une affaire de goût, d'affinité et de compatibilité avec sa propre mécanique cérébrale. J'ai longtemps cru que Verilog était plus adapté pour un développeur, car très sensiblement plus « teinté » programmation. Mais en réalité, ceci était une fausse bonne idée. Pour moi, il semble qu'une distinction brutale soit nécessaire entre le code (comprendre « pour MCU ») et la description de circuits. Ceci sera peut-être différent pour vous, puisque chacun doit faire en fonction de ses propres biais et automatismes parasites, sachant qu'au final, ce qui sera acquis avec un langage pourra être plus facilement traduit/transposé dans l'autre, de toute façon. J'ai essayé d'apprendre avec Verilog à plusieurs reprises, ça n'a pas pris. Ce sera donc du VHDL ici (et comme l'article existe, c'est que ça a marché).

Ah, j'oubliais, il est aussi important de noter que rien ne vous empêche de mélanger VHDL et Verilog et/ou d'utiliser certains éléments (*IP Cores*) écrits dans un langage alors que votre projet en utilise un autre.

surprise (Quartus II Lite 23.1 c'est ~6 Gio, l'IDE GOWIN un peu moins de 1 Gio). Et pour finir sur ce point, dans le cas très particulier d'Altera Quartus, toutes les versions de l'environnement ne supportent pas (ou plus) l'ensemble des FPGA du constructeur. Le Cyclone II, par exemple, nécessitera une version 13.0.1 **maximum**, car il n'y a plus de support pour ce composant dans les versions supérieures et actuelles. J'avais d'ailleurs traité de la manière de jongler avec les anciennes versions (et dépendances obsolètes) sans saccager son système, grâce à Docker, dans le numéro 43 **[5]**.

Notez que si vous voulez juste découvrir le monde des PLD et du VHDL (ou de Verilog) à peu de frais, vous pouvez également opter pour un CPLD comme le MaxII. On trouve des cartes équipées d'un EPM240T100I5 (240 LE) pour moins de 9 €/pièce, mais ce matériel n'est largement pas suffisant pour quelque chose d'aussi complexe que notre petit projet.

Et pour finir, il est important de remarquer que Quartus II, comme l'environnement GOWIN, est scriptable. En d'autres termes, vous pouvez utiliser l'IDE avec son interface graphique ou gérer/éditer vos fichiers à la main avec votre éditeur préféré (Vi, donc) et automatiser les étapes de la production du *bitstream* avec un `Makefile`. Ceci présente l'avantage de rendre le tout beaucoup plus léger et facile à manipuler, si l'on est déjà coutumier des développements en environnement UNIX (voir avec VSCode, si vous voulez), tout en permettant non seulement d'intégrer des étapes complémentaires (comme l'assemblage du code Z80 ou la compilation C avec SDCC), mais aussi de gérer ses développements avec Git. Notez toutefois que l'appel à l'IDE pour affiner des points de configuration et/ou dissiper un flou technique n'est ni impossible ni à écarter totalement. Avec Quartus, modifier des éléments de configuration dans l'IDE ajuste le contenu du fichier QSF sans tout saccager, même en l'absence de « marqueurs » (*tousse* STM32CubeMX *tousse*).

2. L'EXEMPLE Z80 DE NULLOBJECT

Notre base de travail est récupérable via le dépôt GitHub de Joshua Bassett **[3]**. Celui-ci se compose de trois exemples de projets VHDL utilisables directement avec la carte DE0-Nano : un simple compteur permettant de faire clignoter une des LED de la carte (`counter/`), un exemple d'utilisation du composant SDRAM intégré au DE0-Nano (`sdram/`) et un « ordinateur » à base de Z80 avec ROM et RAM (`z80/`).

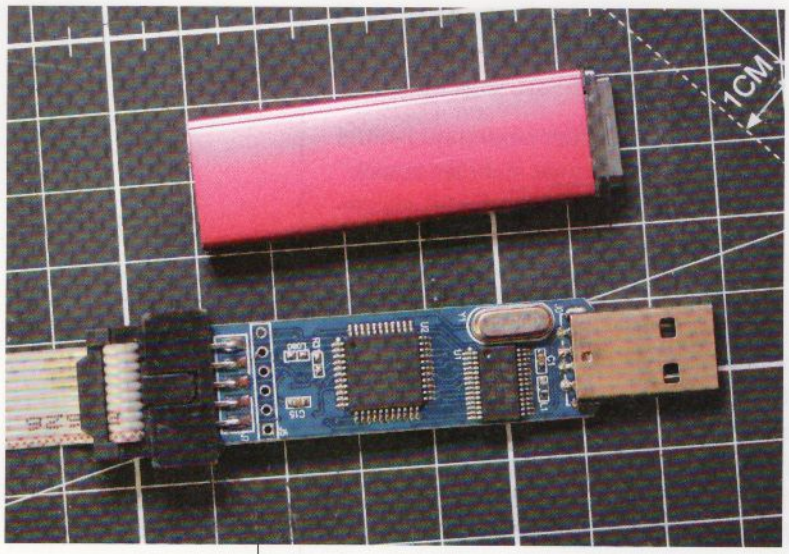
Ce dernier exemple contient cinq fichiers VDHL :

- `clock_divider.vhd` : un générateur d'impulsion connecté au Z80 cadençant les cycles du processeur, basé sur une division de l'horloge principale de la carte à 50 MHz ;
- `reset_gen.vhd` : un circuit permettant à la fois un *reset* automatique à la mise sous tension et un *reset* manuel via un bouton ;
- `single_port_ram.vhd` : la RAM basée sur l'utilisation d'une *megafunction* propre à Altera ;
- `single_port_rom.vhd` : la même chose pour la ROM, prenant en compte un binaire représentant le contenu, issu d'un code assembleur (placé dans `rom/blink.asm`) et à utiliser avec z80asm **[6]** pour produire un fichier binaire qui devra être traduit en format MIF (*Memory Initialization File*), spécifique à Altera, avec `srec_cat` (fourni par le paquet `srecord` sous Debian).
- `top.vhd` : le fichier contenant l'entité principale du projet, liant les autres composants entre eux ou plus exactement les instances de composants.

En plus de cela, on retrouve, dans `lib/`, le *softcore* T80 **[7]**, une implémentation VHDL du mythique Zilog Z80. Notez que ce code est intégré et disponible à la fois sous la forme d'une tripotée de fichiers `.vhd` et d'un fichier `T80.qip`, encore une fois spécifique à Altera. Ce QIP (*Quartus Prime IP*) n'est en réalité qu'un bloc de texte ASCII listant les fichiers VHDL à utiliser. C'est donc directement transposable à d'autres FPGA et environnements, contrairement à la ROM et la RAM.

Ce qui nous amène obligatoirement à parler du jargon en lien avec le domaine. Tout d'abord « softcore » ou plus exactement « processeur softcore » (puisque Wikipédia ne parle pas vraiment de FPGA pour le terme seul) désigne un CPU

Certains devkits intègrent une interface de programmation pour le FPGA. Lorsque ce n'est pas le cas, il faut avoir recours à un programmeur comme cet USB Blaster (probablement un clone).



Parfois, les clones de programmeurs USB Blaster sont facilement identifiables par leur aspect et leur médiocre qualité. Celui-ci n'a clairement aucune chance de passer pour le produit original, mais ceci dit, il fonctionne très bien et ne coûte que 6 €.

implémenté dans un circuit logique programmable (PLD) comme un FPGA. Ce *softcore* est également un « IP Core », réduction de « *Intellectual Property Core* », désignant le design d'un circuit intégré, qu'il soit configuré dans un PLD ou « fondu » dans un ASIC (*Application-Specific Integrated Circuit*), un circuit intégré spécialisé créé pour une tâche précise. La notion de « *Intellectual Property* » est liée au fait qu'il s'agit d'une « conception », répondant donc aux critères d'une propriété intellectuelle (comme une œuvre musicale, par exemple) et qu'il fait l'objet de vente de licences d'utilisation. C'est par exemple ce que fait ARM, qui ne fabrique physiquement rien, mais conçoit des *IP Cores* dont les licences sont vendues à des sociétés comme Broadcom, NVIDIA, Apple, Samsung ou encore Qualcomm, qui produit les circuits intégrés.

Dans le monde d'Altera, un certain nombre d'*IP Cores* gratuitement utilisables et paramétrables sont fournies sous la forme de bibliothèques, ce sont les *megafunctions*. Le projet de nullobject en utilise une, appelée « altsyncram », pour créer la RAM et la ROM de l'ordinateur en utilisant les blocs mémoire intégrés au FPGA (blocs M9K). Ceci ne sera pas transposable directement sur un autre FPGA et il sera alors nécessaire de trouver un *IP Core* similaire dans le nouvel environnement (chez GOWIN, c'est « pROM » et « SP »). Il est également possible de trouver des *IP Cores* non seulement gratuits, mais *open source*, par exemple via le site OpenCores **[8]**. C'est d'ailleurs ce que j'ai fait pour ajouter une UART dans le projet final. Notez que ces *IP Cores* simples comme des blocs mémoire, des contrôleurs de bus i2c ou SPI, ou encore des interfaces diverses sont souvent utilisables gratuitement (sous conditions), mais que des choses plus complexes comme une interface PCIe ou un CPU *softcore* nécessitent l'achat de licences (parfois après une période d'essai) relativement coûteuses.

Accompagnant tout cela, nous trouvons également deux autres fichiers qui concernent le projet lui-même, à commencer par le `Makefile`, permettant de produire le *bitstream* pour configurer le FPGA et de le programmer dans le composant via l'USB Blaster. Ce sont les outils en ligne de commande Quartus qui sont appelés et l'auteur suppose qu'ils se trouvent dans le `$PATH` (tout comme `z80asm` et `srec_cat`). Ceci ne sera généralement pas le cas et on ajustera en fonction du répertoire d'installation de la suite logiciel Altera. Ces outils ont heureusement le bon ton de pouvoir être utilisés de n'importe où en fournissant le chemin absolu complet. Notez que le `Makefile` est relativement basique

(pour dire les choses gentiment) et supprimera les sources assembleur (tout le répertoire `rom/`) en cas de `make clean`.

Le second fichier est `z80.qsf`, pour *Quartus Settings File*, et contient l'ensemble de la configuration du projet. On y trouve le type et le modèle de FPGA utilisé, la correspondance entre les broches du composant et les signaux utilisés dans le VHDL, la liste des fichiers qui composent le projet et quelques autres éléments de configuration. Notez qu'un fichier `z80.qpf` (*Quartus Project File*) est également présent et pourra être ouvert avec Quartus, en mode graphique/IDE.

Si vous disposez d'un DE0-Nano, et que vous avez installé les outils complémentaires, vous pouvez tout simplement utiliser la commande `make build` puis `make program` pour synthétiser le tout et programmer le FPGA. Vous devriez alors voir les huit LED de la carte clignoter. Attention, ceci n'est pas un simple *blink* en VHDL, c'est le code machine en ROM, exécuté par le *softcore* Z80, qui utilise l'instruction `OUT`. Le tout à partir d'un code assembleur (`blink.asm`) utilisant l'instruction `CALL` pour la temporisation, et donc avec une sous-routine et une utilisation de la pile (*stack*), prouvant ainsi également que la RAM fonctionne très bien. À mon sens, c'est déjà impressionnant et fascinant ! Merci, Joshua.

2.1 Tour du propriétaire : le code VHDL

Nous allons ici surtout nous intéresser au contenu de `top.vhd` et donc à l'entité `top` qu'elle implémente. Dans un langage de description de matériel ou HDL (*Hardware Description Language*), on décrit un circuit, on ne programme pas. Pensez HTML, ne pensez pas C, Java, Python, etc. Pour structurer notre description, nous avons besoin d'un point de départ, quelque chose qui va détailler ce qui est présent et la façon dont les éléments sont connectés et réagissent entre eux. C'est l'entité racine de tout le design et celle que vous allez synthétiser (ou simuler).

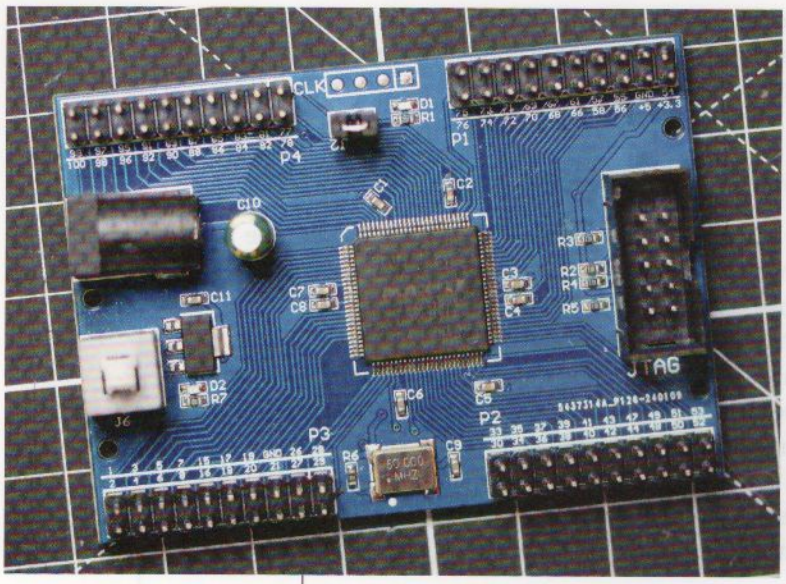
D'autres entités (ou « modèles » en bon français, à priori) sont également présentes, comme le *softcore* T80, la RAM, la ROM, le générateur d'impulsions, le circuit de *reset*, etc. Et elles-mêmes peuvent être composées d'instances d'autres entités. Une entité UART, par exemple, peut reposer sur des FIFO, un générateur d'horloge, un circuit RX et TX, etc. Si une entité est présente dans le code VHDL (ou Verilog, peu importe), mais non liée au reste de ce qui est, de fait, une arborescence, elle ne sera pas synthétisée.

De plus, une entité peut être utilisée plusieurs fois dans un projet, deux UART par exemple ou encore plusieurs diviseurs de fréquences. L'entité elle-même n'est qu'un plan, un « pochoir », pour créer un morceau de circuit. On n'utilise donc pas réellement l'entité elle-même, mais plus exactement une ou des **instances** de cette entité.

Si vous jetez un œil au fichier `z80.qsf`, vous verrez une ligne `TOP_LEVEL_ENTITY` contenant `top`, et dans `src/top.vhd` nous avons :

```
entity top is
[...]
end top;

architecture arch of top is
[...]
begin
[...]
end arch;
```



La famille des circuits logiques programmables ne se limite pas aux FPGA et on trouve, pour très peu cher (< 10 €), ce type de carte reposant sur un CPLD MaxII EPM240. Très bien pour expérimenter à petite échelle, mais insuffisant pour quelque chose d'aussi complexe que notre projet.

Note : avant toute chose, je précise que je pars maintenant du principe que vous avez le code VHDL en question sous les yeux et vous vous y référez en parallèle de la lecture de ce qui suit. Ceci m'évitera d'encombrer inutilement l'article de centaines de lignes assez répétitives.

Cette entité `top` sera automatiquement instanciée par l'environnement, car c'est l'entité racine (oui, le terme `TOP_LEVEL_ENTITY` est un peu contre-intuitif). Sa description se divise en deux parties avec tout d'abord la déclaration de l'entité listant les signaux en entrée et en sortie (liés à des broches physiques précisées dans le fichier QSF), puis son architecture contenant les signaux internes et des descriptions de circuits, incluant des instanciations d'autres entités. On pourrait voir cela comme une fonction, avec arguments et corps, mais je pense que c'est une image plus parasite qu'autre chose.

Dans l'implémentation actuelle, nous avons donc la déclaration suivante :

```
entity top is
  port (
    clk : in std_logic;
    key : in std_logic_vector(1 downto 0);
    led : out std_logic_vector(7 downto 0)
  );
end top;
```

Notre `top` utilise trois `port` (qu'on peut littéralement imaginer comme les pattes d'un composant) :

- `clk` qui est le signal d'horloge de 50 MHz arrivant (`in`) par la broche `PIN_R8` sur DE0-Nano ;
- `key` qui est un vecteur de 2 lignes correspondant respectivement à `PIN_J15` (`key[0]`) et `PIN_E1` (`key[1]`) ;
- et `led` un autre vecteur, en sortie cette fois (`out`), connecté aux 8 broches des LED intégrées à la carte (cf. le QSF pour les broches).

Ceci représente, en gros, la connexion de notre entité avec le monde extérieur. Cette déclaration d'entité est ensuite suivie par son architecture, qui se divise en deux parties avec d'une part la « zone déclarative » :

```
architecture arch of top is
  -- clock enable
  signal cen : std_logic;

  -- cpu reset
  signal reset : std_logic;

  -- address bus
  signal cpu_addr : unsigned(15 downto 0);

  -- data bus
  signal cpu_din  : std_logic_vector(7 downto 0);
  signal cpu_dout : std_logic_vector(7 downto 0);
[...]
```

Puis, suit le bloc où se trouve la description fonctionnelle de l'entité, ou en d'autres termes, ce qu'elle « fait » (ou « est ») :

```
begin
  clock_divider : entity work.clock_divider
  generic map (DIVISOR => 50)
  port map (clk => clk, cen => cen);
[...]
end arch;
```

Plusieurs remarques s'imposent immédiatement. Premièrement, les lignes débutant par un double tiret sont des commentaires (comme `//` en C). Il n'y a **pas** de syntaxe pour des commentaires multilignes (`/*...*/`) en VHDL, ne cherchez pas. Précisons aussi que VHDL ne fait pas de distinction entre majuscule et minuscules. `std_logic`, `STD_LOGIC`, `STD_logic` ou `StD_LoGIc` reviennent au même.

Ensuite, dans la déclaration, les signaux sont séparés par des points-virgules et, de ce fait, la dernière ligne (`led`) n'en possède pas. Ceci n'est pas vrai dans la zone déclarative du bloc de description de l'architecture de l'entité (c'est perturbant, d'autant que par ailleurs, c'est une virgule qui est utilisée comme séparateur, la logique m'échappe). Le réflexe du développeur C, consistant à coller des « `;` » partout, doit donc être refréné.

Viennent ensuite les types pour les signaux et il en existe une petite collection. Je ne vais pas tous les lister ici, mais nous avons dans `top.vhd` :

- `std_logic` qui est un simple signal logique pouvant être `X` inconnu, `0` état logique bas, `1` état logique haut et `Z` haute impédance (ouvert).
- `std_logic_vector` qui est une déclinaison de `std_logic` avec plusieurs lignes (un bus donc). L'argument passé précise la taille, avec par exemple `7 downto 0` pour huit bits avec le bit de poids fort à gauche (`0 to 7` pour le sens inverse (non, ce n'est pas `upto`)).

- `unsigned` est un vecteur de bits comme `std_logic_vector`, avec plusieurs lignes donc, mais celui-ci permet de faire des opérations arithmétiques, car les bits sont traités comme ceux d'un entier non signé. Il existe également `signed`, équivalent signé. Inversement, un `unsigned` ou un `signed` est un entier qui ne supporte pas les opérations logiques.

Tous ces types, en réalité, n'existent pas en VHDL et proviennent de *packages* de la bibliothèque standard IEEE qui sont spécifiés en début de `top.vhd` :

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

Point important, les ports de notre entité (`clk`, `key` et `led`) sont aussi automatiquement des signaux dans l'architecture. Nous n'avons pas besoin de les redéclarer dans la zone déclarative (un peu comme des variables en argument d'une déclaration de fonction en C existent dans le corps de la fonction).

Enfin, dans le court extrait précédent, nous avons une instanciation d'entité, celle de `clock_divider`, le générateur d'impulsions pour le *softcore* Z80. Ces trois lignes constituent ce qu'on appelle une instanciation directe (par opposition à l'instanciation par composant, que nous laisserons de côté pour le moment). Mais prenons un exemple plus complet, celui de la ROM :

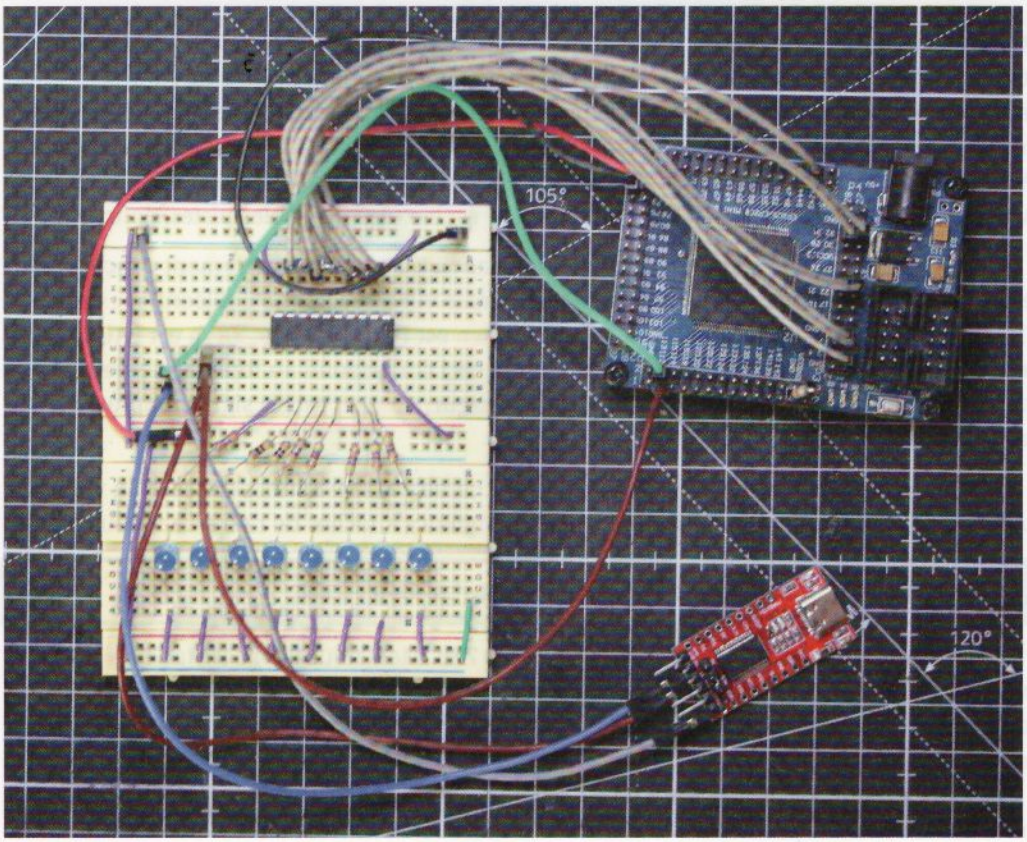
```
  rom : entity work.single_port_rom
  generic map(
    ADDR_WIDTH => 12,
    DATA_WIDTH => 8,
    INIT_FILE  => "rom/blink.mif"
  )
  port map(
    clk  => clk,
    cs   => rom_cs and not cpu_mreq_n and not cpu_rd_n,
    addr => cpu_addr(11 downto 0),
    dout => rom_dout
  );
```

L'entité concernée est `single_port_rom` (avec `work` désignant la bibliothèque courante durant l'analyse, en gros, nous-mêmes) provenant du fichier `single_port_rom.vhd` (les deux noms ne sont pas liés, vous pouvez appeler vos fichiers et entités comme vous voulez) où l'on voit :

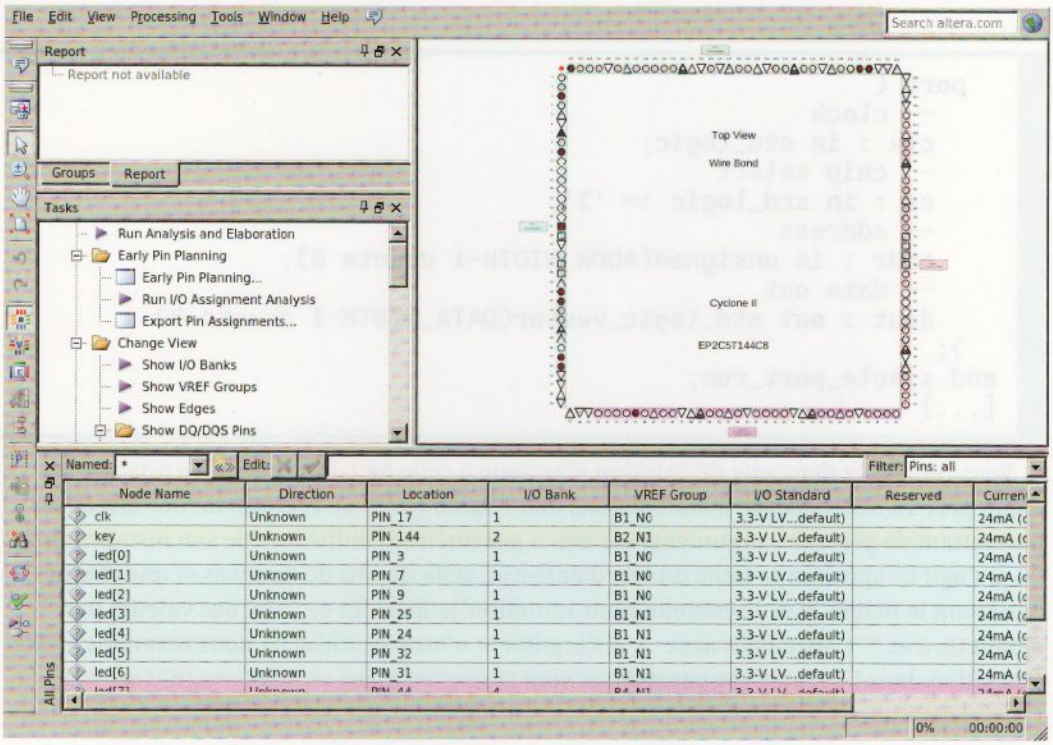
```
entity single_port_rom is
  generic (
    ADDR_WIDTH : natural := 8;
    DATA_WIDTH : natural := 8;
    INIT_FILE  : string := ""
  );
```

```
  port (
    -- clock
    clk : in std_logic;
    -- chip select
    cs : in std_logic := '1';
    -- address
    addr : in unsigned(ADDR_WIDTH-1 downto 0);
    -- data out
    dout : out std_logic_vector(DATA_WIDTH-1 downto 0)
  );
end single_port_rom;
[...]
```

Vous retrouvez donc une déclaration identique à celle de `top` à une petite nuance, ou ajout, prêt. Avant les ports, nous trouvons une section `generic` regroupant des directives permettant de gérer des arguments servant à paramétrer l'entité lors de son instanciation. Ici, il s'agit de spécifier la taille du bus d'adresses, celle du bus de données et une chaîne désignant le fichier avec le contenu pour initialiser la ROM. Et ce, avec des valeurs par défaut (`8`, `8` et `""`), spécifiées avec `:=` si l'argument n'est pas utilisé. Si nous revenons à l'instanciation dans `top.vhd`, ces paramètres sont respectivement `12`, `8`, et `rom/blink.mif`,



Voici le montage « de base » à l'origine de cet article. La carte en haut à droite repose sur un FPGA Altera Cyclone II obsolète, mais qu'il est toujours possible de trouver en ligne [15]. C'est une solution économique (~20 €) pour découvrir les FPGA Cyclone et leur environnement.



L'environnement de développement Quartus II permet de faciliter la configuration des broches du composant via une interface graphique. Les réglages appliqués auront pour effet de modifier le fichier QSF, que vous pourrez ensuite ajuster selon vos besoins.

et passés via la directive `generic map`. De la même manière, les ports de l'entité seront connectés à des signaux internes à `top` via `port map`, avec le nom du port de l'entité à gauche et celui du signal connecté à droite (et, oui, les noms peuvent parfaitement être identiques, ce qui ne facilite pas la lecture, cf. l'instanciation de `clock_divider`). Ceci revient littéralement à brancher les ports de l'instance de `single_port_rom` (que nous appelons `rom`) aux signaux que nous avons déclarés dans l'entité racine (des fils/câbles/pistes, donc).

Deux choses sont intéressantes dans cette instanciation et plus exactement concernant les ports. `cs` est connecté, non pas directement, mais via une logique booléenne. `cs` est VRAI (état haut) si `rom_cs` est VRAI, ET `cpu_mreq_n` est FAUX, ET `cpu_rd_n` est FAUX. Si vous avez suivi les articles sur le Z80 sur platine, ceci correspond au fonctionnement normal avec les signaux /RD et /MREQ du vrai processeur. `rom_cs` est un signal en plus, géré par ailleurs pour distinguer RAM et ROM.

Le second point concerne le nombre de lignes ou bits des vecteurs utilisés. Non visible dans mon extrait, mais présent dans la liste des signaux internes de notre `top`, nous avons :

```
signal cpu_addr : unsigned(15 downto 0);
```

Un bus de 16 bits d'adresses classiques pour le Z80. Mais nous avons instancié `rom` avec `ADDR_WIDTH => 12`, c'est une mémoire de seulement 4 Kio (2^12 = 4096). Pour la connecter au bus 16 bits, nous devons donc spécifier quelles lignes sont concernées. C'est l'objet de la syntaxe `cpu_addr(11 downto 0)` signifiant « les bits 11 à 0 », sous-entendu « sur les 16 existants ». Nous pouvons ainsi connecter des ports et des signaux d'une taille différente sans problème.

2.2 Que fait le code de nullobject ?

À présent que nous avons quelques informations concernant la structure et la syntaxe d'un code VHDL, nous pouvons comprendre ce qu'implémente ce projet. Comme nous l'avons vu, l'entité `top` est connectée, via ses ports, à des broches physiques du FPGA Cyclone IV : horloge, LED et boutons. En termes de signaux internes à l'architecture, nous retrouvons (de la ligne 50 à 90) les bus et signaux classiques d'une machine à base de Z80 : bus d'adresses, bus de données (au pluriel), signaux en sortie (/MREQ, /IORQ, /RD, /WE), signaux en entrée (`reset`) et de quoi activer (/CS) la ROM, la RAM et un registre pour les LED.

Soulignons que ce sont des signaux, littéralement des fils, qui, si on s'en tient à cela, ne sont connectés à rien pour l'instant. On se rend également compte que tout ceci n'est pas totalement identique à une configuration physique où nous avons un unique bus de données. Ici, nous avons une distinction entre les données en entrée (du CPU) et celles en sortie, respectivement `cpu_din` et `cpu_dout` (un vrai Z80 n'est pas broché ainsi). Il en va de même pour la ROM et la RAM avec `ram_dout`/`rom_dout` et ceci se constate également dans l'instanciation du *softcore* (T80) et de ces entités. Leurs ports distinguent entrées et sorties :

- `DI` et `DO` pour le CPU ;
- `din` et `dout` pour la RAM ;
- et `dout` pour la ROM (pas de `din`, on n'écrit pas dans une ROM).

Les ports d'une entité peuvent être en mode `in` ou `out` comme nous l'avons vu précédemment. Du point de vue d'une entité elle-même, un port `in` peut être lu par son architecture, mais pas écrit, et inversement, `out` peut être écrit, mais pas lu. De l'extérieur de l'entité, c'est l'inverse, un port `in` est quelque chose à quoi on impose un état, et `out` d'où on obtient l'état. D'autres modes existent, comme `inout`, mais ceci complique grandement les choses, car comme dans un vrai circuit, un seul composant peut imposer un état, les autres doivent être déconnectés (signal à l'état « Z », ouvert). La plupart des entités, autrement dit des composants *IP Cores*, que vous pourrez trouver séparent généralement données en entrée et en sortie comme ici avec le T80 et la ROM/RAM, mais aussi avec les *softcores* T65 (6501) et fx68k (68000), par exemple.

En dehors des instanciations d'entités, le reste du VHDL se résume à peu de choses. Nous avons, dans le désordre :

```
-- mux CPU data input
cpu_din <= rom_dout or ram_dout;

rom_cs <= '1' when cpu_addr >= x"0000"
          and cpu_addr <= x"0fff" else '0';
ram_cs <= '1' when cpu_addr >= x"1000"
          and cpu_addr <= x"1fff" else '0';
led_cs <= '1' when cpu_addr(7 downto 0) =
          x"00" else '0';

led <= led_reg;
```

Nous avons là des assignations simples où le résultat d'une expression est assigné à un signal. La première assignation est évidente puisque l'état de `cpu_din` découle d'une opération logique. Si `rom_dout` OU `ram_dout` est VRAI, alors `cpu_din` est vrai. Les trois autres permettent de faire exactement la même chose que ce que ferait un circuit logique avec un Z80 sur platine à essais : activer l'une ou l'autre ligne /CS des composants en fonction de l'état des lignes sur le bus d'adresses. Ici, si l'adresse est entre 0x0000 et 0x0fff, nous sélectionnons la ROM, si c'est entre 0x1000 et 0x1fff, ce sera la RAM et si les huit bits de poids faible sont à 0x00, c'est `led_cs` qui est activé. Notez que, dans ces trois lignes, le premier `<=` est une assignation, mais le second est une condition « inférieure ou égale » (perturbant).

Si vous connaissez le Z80, vous aurez deviné que `led_cs` et `led_reg` correspondent finalement à un composant adressé lorsque /IORQ est à l'état bas et répondant donc à l'instruction assembleur `OUT`. Mais nous n'avons nulle trace de `cpu_ioreq_n`, alors que `cpu_mreq_n` est utilisé pour l'instanciation de la ROM et de la RAM. L'auteur du code gère cela autrement, avec le dernier morceau de code qui nous reste à voir, un *process* (je laisse la traduction pertinente à vos bons soins, bonne chance) :

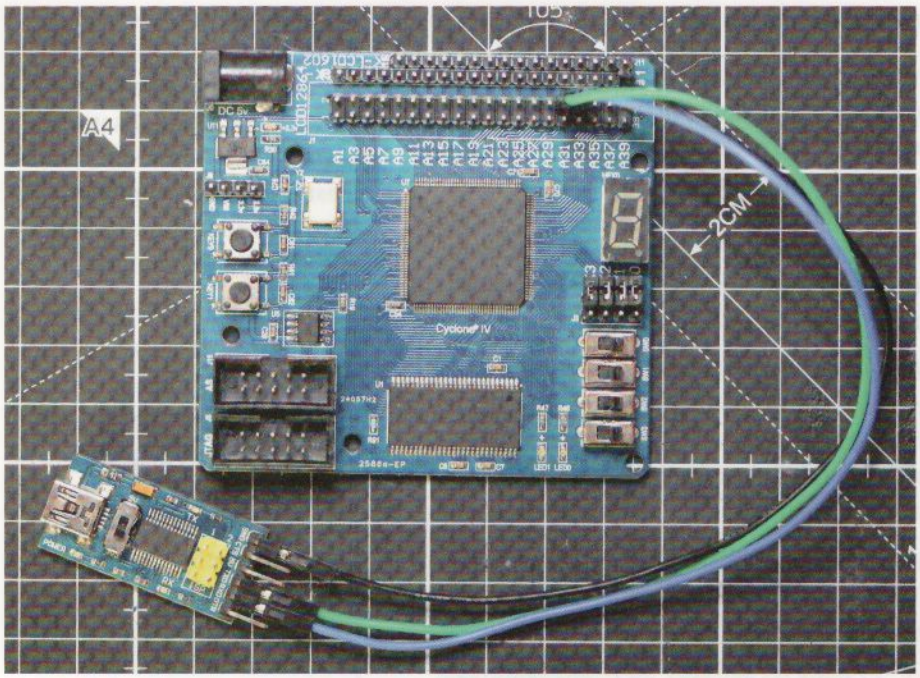
```
set_led_register : process (clk)
begin
  if rising_edge(clk) then
    if led_cs = '1' and cpu_ioreq_n = '0'
        and cpu_wr_n = '0' then
      led_reg <= cpu_dout;
    end if;
  end if;
end process;
```

Le morceau de code précédent était un ensemble d'assignations concurrentes, ce qui veut dire que l'état de `rom_cs` ne change pas après `cpu_din` et/ou avant `led_cs`, mais **en même temps**. L'ordre du code ne compte pas.

Avec un *process* c'est différent (un peu), car les instructions qui s'y trouvent sont séquentielles, même si le *process* lui-même est concurrent à d'éventuels autres *process* et aux assignations concurrentes. Nous avons ici un *process* appelé `set_led_register` et est précisé entre parenthèses sur la première ligne sa « liste de sensibilité » ou *sensitivity* en anglais (plus pertinent, je trouve). Le *process* est « sensible » à un changement du signal `clk` qui, je le rappelle, est lié à l'oscillateur de 50 MHz en entrée de notre entité racine. Il est possible de préciser plusieurs signaux dans la sensibilité d'un *process* en utilisant une virgule, il va alors réagir sur n'importe lequel de ces signaux. Lorsqu'une sensibilité est utilisée, le *process* est suspendu lorsque les instructions qu'il contient ont été exécutées une à une.

Nous avons donc ici quelque chose de déclenché par un changement de `clk` et dans le corps du *process*, nous avons immédiatement un test pour déterminer le type de changement : `if rising_edge(clk) then`. Nous agissons en cas de front montant sur `clk` (état bas vers haut). Là, nous testons si `led_cs` est VRAI, `cpu_ioreq_n` est FAUX (c'est /IORQ) et `cpu_wr_n` est FAUX (c'est /WR). Si c'est le cas, l'état de `cpu_dout`, les huit bits de données en sortie du *softcore*, est assigné à `led_reg`.

Comme nous avons une assignation simple et concurrente de `led_reg` à `led`, les bits présentés sur le bus de données sont directement « affichés » sur les LED. `led_reg` est donc



Cette carte, basée sur un Cyclone IV, est un véritable cauchemar : nombre de broches disponibles très limité et absolument aucune documentation. Le projet a fini par fonctionner avec ce devkit, mais il a fallu passer par une très longue et très pénible étape de rétro-ingénierie au testeur de continuité. Quelle que soit la carte que vous choisissez, assurez-vous toujours d'avoir, au minimum, accès au schéma du circuit avant achat.

techniquement, comme son nom l'indique, un registre, dont l'état change lorsque le code assembleur exécuté par le Z80 utilise l'instruction `OUT` (/IORQ + /WR) à l'adresse 0x00 (qui provoque le passage de `led_cs` à 1, VRAI).

Les conditions comme `if`, `then`, `else`, mais aussi `case`/`when` (équivalent à un `switch`/`case`), ou `for`/`generate`, ne peuvent être utilisées **que** dans un *process*. Autre point important, sinon critique, l'assignation d'un état (ou « affectation d'un signal ») n'arrive **qu'une fois**. Si on assigne plusieurs états consécutifs à un signal dans un *process*, seule la dernière assignation comptera. Et celle-ci n'est effective qu'à la ligne `end process`. Même dans un *process*, un signal est un câble, **pas une variable**. Et un *process* n'est **pas un code** exécuté par le FPGA, mais une simple facilité pour décrire le comportement d'un circuit.

3. AJOUTONS NOS BRIQUES ET RÉVISONS NOTRE COPIE

Notez que je ne vais pas lister, dans cet article, le contenu de tous les fichiers ajoutés, car le magazine n'aurait pas assez de pages. Ce projet, dans l'état où il se trouvera bien après la fin de la rédaction (en principe donc plus affiné encore), pourra être téléchargé et étudié directement depuis un de mes dépôts GitLab **[9]**. Si quelque chose vous paraît flou ou semble faire l'impasse sur un point, vous trouverez la réponse à cet endroit, directement dans le code et les fichiers de configuration (et, oui, mes commentaires sont toujours en anglais dans un code public et libre).

3.1 Réorganisation du projet

Avant de nous lancer dans la modification du VHDL et l'ajout de fonctionnalités, un brin de ménage s'impose. Le `Makefile`, en particulier, est excessivement simpliste et inadapté à quelque chose de plus conséquent qu'une simple démonstration. Mais avant cela, nous allons ajuster l'arborescence et accessoirement rendre le tout plus facilement adaptable à différents modèles de FPGA, cartes et *devkits*. Nous aurons donc, comme sous-répertoires :

- `rtl/` (pour *Register Transfer Level*, la description de l'architecture d'un circuit) pour les fichiers qui se trouvent actuellement dans `src/`, à l'exception de `top.vhd`. Il s'agit là des éléments invariables d'une carte à une autre ;
- `lib/` pour les éléments « externes » comme les fichiers du *softcore* T80 et sous peu ceux de l'UART que nous allons ajouter ;
- `romsrc_uart_c/` pour les sources, en C, qui remplaceront l'actuel code assembleur et dont le résultat binaire ira en ROM ;
- `boards/` qui contiendra une arborescence avec un répertoire par modèle de cartes supportées (`de0nano/` en premier lieu), qui contiendra le `Makefile`, le fichier QSF, les fichiers de configuration pour générer, convertir et programmer le *bitstream* et, dans un sous-répertoire `rtl/`, notre `top.vhd` adapté au FPGA concerné.

En VHDL, il n'y a pas de `include` ou de macros, et bien qu'il existe des techniques pour rendre un code paramétrable, il est bien plus facile dans notre cas relativement simple de maintenir plusieurs `top.vhd` en parallèle. Cette adaptation est nécessaire à cause de la RAM/ROM, car chaque modèle de FPGA dispose de plus ou moins de mémoire interne utilisable et le comportement du circuit doit s'y adapter (*mapping* mémoire, etc.). Cette problématique impactant également le code source en C, compilé avec SDCC comme nous l'avons vu avec de précédents articles, une approche sensiblement différente est utilisée. L'ensemble des sources se trouve dans `romsrc_uart_c/`, mais nous aurons plusieurs `Makefile` (`Makefile.8kB_4kB`, `Makefile.32kB_16kB`, etc.) et plusieurs *crt0* (`mycrt0_8_4kB.s`, `mycrt0_32_16kB.s`, etc.), en fonction du volume de la ROM et de la RAM, et de la division des adresses entre les deux (ainsi que l'initialisation du pointeur de pile). La source C est la même, l'initialisation et la compilation sont différentes. Il est certainement possible de faire plus concis, mais pour l'heure, cela fera l'affaire.

Un autre point important, à mon sens, est de rendre la construction (et la compilation du C) conditionnelle, puisque c'est l'un des atouts majeurs de l'utilisation de `make`. N'importe quel changement dans le code VHDL, la configuration ou le code C/assembleur doit déclencher une reconstruction, mais pas dans le cas contraire. Nous en profiterons d'ailleurs pour séparer les étapes de la création du *bitstream*, car le `Makefile` d'origine utilise simplement `quartus_sh`, le *Quartus Prime Shell*, sorte d'interface unique en ligne de commande. Nous, nous allons diviser cela en :

- l'analyse / élaboration avec le *mapper* `quartus_map` ;
- le placement et le routage avec le *fitter* `quartus_fit` ;
- et l'assemblage (au sens « fusion » de l'ensemble, pas « langage ») avec l'assembleur `quartus_asm`.

À chaque phase, nous obtenons un certain nombre de fichiers dans `output_files/`, nécessaires pour l'étape suivante, jusqu'à arriver à la création d'un SOF (*SRAM Object File*) qui peut être utilisé pour

configurer le FPGA Cyclone. Mais attention, cette configuration, qui permet effectivement de faire fonctionner notre circuit dans le composant, est volatile. Une coupure d'alimentation nous fera perdre notre résultat.

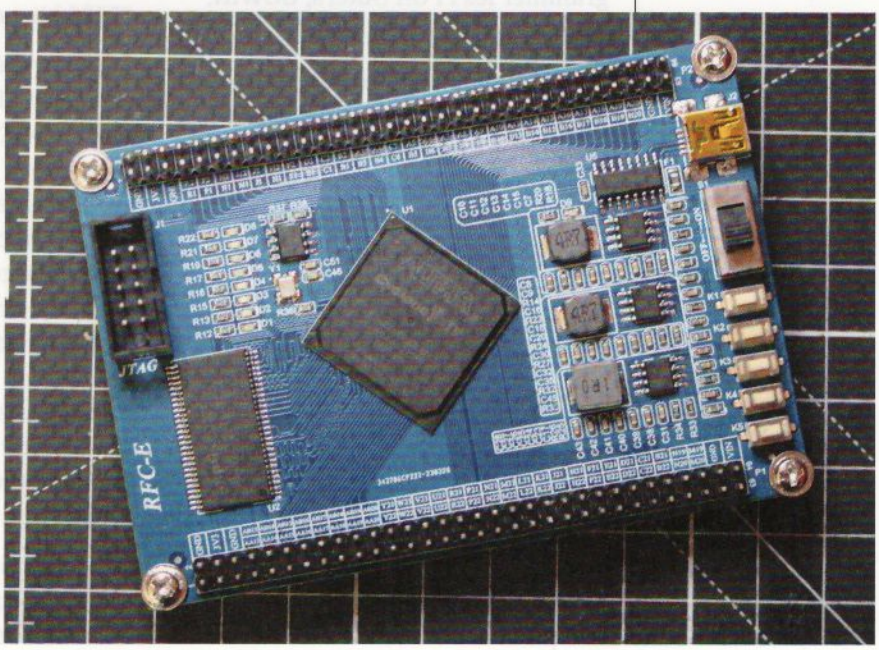
Pour rendre tout cela plus permanent (mais non définitif), les FPGA Cyclone sont accompagnés d'une flash NOR interfacée en SPI qui leur permet de lire leur configuration et de l'activer à la mise sous tension. Ce sont des composants qui, chez Altera/Intel, sont appelés EPCS (*Erasable Programmable Configurable Serial*) : EPCS1, EPCS4, EPCS16, EPCS64... selon le volume de données supporté (en bits), 1 Mb, 4 Mb, 16 Mb, etc.

Et c'est là que les choses se compliquent un peu, car pour programmer cette mémoire, nous devons passer par le FPGA. Il existe d'autres façons de faire, mais celle-ci est celle par défaut. Techniquement, un *bitstream* particulier est configuré dans le FPGA pour qu'il serve d'interface et c'est lui qui réceptionne alors nos données via le port JTAG et les inscrit en flash. En pratique, l'outil de programmation, `quartus_pgm`, fait cela à votre place, soit via des options en ligne de commande, soit via un petit fichier de configuration (CDF pour *Chain Description File*), mais en utilisant le *bitstream* dans un format particulier, JIC (pour *JTAG Indirect Configuration*).

À ce stade, vous vous dites sans doute qu'Altera souffre d'une sorte d'obsession pour les acronymes de trois lettres (comme IBM, fut un temps) et vous avez peut-être raison, car ce n'est pas fini. Pour obtenir un JIC à partir de notre SOF, nous devons le convertir avec `quartus_cpf`, en utilisant un fichier de configuration ou COF pour *COnversion setup File* (celui-ci est un peu « limite », tout comme CPF qui signifie « *Quartus Prime Convert_programming_file* »). Au final, nous devons donc compléter le `z80.qsf` avec les fichiers :

- `z80_jic.cof` pour transformer le SOF en JIC ;
- `z80_SOF.cdf` pour charger la configuration dans le FPGA de manière volatile ;
- `z80_EPCS64.cdf` pour charger la configuration dans la flash (EPCS64 sur DE0-Nano).

Vous retrouverez ces fichiers (simple ASCII) dans le dépôt GitLab, mais pouvez également les créer en utilisant l'IDE Quartus, via ***File*** et ***Convert Programming Files*** et ***Tools*** puis ***Programmer***,



Ce kit Cyclone IV de PISwords [16] coûte une cinquantaine d'euros, mais comprend un EP4CE15F23C8 (15408 LE, 4 PLL, 344 broches E/S et 516096 bits de SRAM), huit LED, une interface USB/série et 5 boutons-poussoirs. Rien n'a donc besoin d'être ajouté pour faire fonctionner notre projet (si ce n'est le programmeur).

Les cartes FPGA et autres devkits intègrent parfois de la mémoire supplémentaire sous la forme d'une SDRAM (ici, de 256 Mb) qui permet, moyennant développement d'un support adéquat, de s'affranchir des limitations de la SRAM intégrée au FPGA.

respectivement pour la conversion et la programmation. Dans les interfaces qui s'affichent, vous trouverez une option permettant de stocker la configuration choisie dans un fichier (COF et CDF). On pourra éventuellement, et selon le modèle de FPGA, ajouter un COF pour la transformation en RBF (*Raw Binary File*) pour une utilisation avec openFPGALoader **[10]**, un outil *open source* polyvalent pour programmer les FPGA (Altera, GOWIN, Xilinx, Lattice, Efinix, etc.).

Concernant à nouveau la source en C et le binaire devant prendre place en ROM, là aussi nous devons procéder à une conversion. Le `Makefile`, spécifique à chaque organisation mémoire, ne fait que produire un `.bin`, qui pourrait parfaitement être programmé dans une vraie EEPROM. Pour l'utiliser avec le projet et un FPGA Altera, nous le transformerons en un `.mif`. Ceci se fera directement depuis le `Makefile` du sous-répertoire dans `board/` avec `srec_cat` (option `-mif`) en précisant un fichier de sortie `romcode.mif` dans `output_files/` afin que Quartus le trouve (et non plus dans `rom/`).

Et enfin, puisque nous avons déplacé pas mal de fichiers, il conviendra de faire un tour dans `z80.qsf` pour ajuster les chemins. Notez que des emplacements relatifs fonctionnent tout aussi bien que des chemins absolus. Et ceci est valable également pour les fichiers COF et CDF (Quartus a une fâcheuse tendance à utiliser des chemins absolus dans la génération de fichiers, ce qui casse tout au moindre déplacement/copie/clone/*pull*).

3.2 Migration vers Cyclone II

À présent que nous avons mis un peu d'ordre dans tout cela et rangé le développement réalisé dans son répertoire `board/de0nano/`, il est temps de transposer tout cela vers un autre FPGA (de la même famille). Le Cyclone IV E du DE0-Nano est un EP4CE22F17C6 disposant de 22320 éléments logiques (LE), 608256 bits de mémoire interne (blocs M9K) et 154 broches. Le projet de nullobject tel qu'il existe actuellement utilise seulement 2350 LE, 65536 bits de mémoire (ROM+RAM = (4092+4096)x8 = 65536 bits) et 11 broches.

La carte économique chinoise à base de Cyclone II (obsolète) repose sur un EP2C5T144C8 avec 4608 LE, 119808 bits de mémoire et 89 broches. Techniquement (et grossièrement), le projet rentre parfaitement dans ce FPGA. Mieux encore, lorsqu'on regarde le contenu de `rtl/single_port_ram.vhd`, nous voyons que l'*IP Core* utilisée est paramétrée avec `intended_device_family => "Cyclone II"`. Passer du DE0-Nano à la carte Cyclone II va drastiquement faire chuter le budget, tout en nous permettant d'obtenir un « ordinateur Z80 » plus complet, que du bonheur.

La description VHDL, bien que sensiblement liée au FPGA (ou la famille de FPGA) en raison de l'*IP Core*, ne nécessite aucune

modification pour le moment. Si nous voulons simplement obtenir la même chose sur Cyclone II, nous n'avons besoin de modifier que le contenu du fichier QSF. L'approche que j'ai choisie consiste à utiliser l'IDE Quartus et créer un nouveau projet fictif/temporaire via ***File*** et ***New Project Wizard***. Là, après une fenêtre d'introduction, on vous demandera de spécifier un répertoire et un nom pour le projet, avant de vous laisser choisir entre un projet vide ou basé sur un *template* et de vous demander quels sont les fichiers à intégrer. Notre projet sera vide et nous n'avons pas de VHDL/Verilog à ajouter (***Next***, donc). L'écran suivant vous permet de choisir le FPGA utilisé (ou la carte pour un *devkit* « connu »). Optez pour la famille « Cyclone II » et sélectionnez le composant dans la longue liste présentée en dessous, puis cliquez ***Finish***. Vous n'avez pas même à enregistrer le projet et pouvez quitter Quartus, le fichier QSF est déjà créé à l'emplacement que vous aurez spécifié. Notez que si la famille « Cyclone II » est absente, soit votre installation de Quartus est incomplète, soit votre version est trop récente (c'est 13.0.1 **maximum** pour un Cyclone II).

Dans le fichier QSF, nous trouvons :

```
set_global_assignment -name FAMILY "Cyclone II"
set_global_assignment -name DEVICE EP2C5T144C8
set_global_assignment -name TOP_LEVEL_ENTITY monproj
set_global_assignment -name ORIGINAL_QUARTUS_VERSION "13.0 SP1"
set_global_assignment -name PROJECT_CREATION_TIME_DATE "14:08:47  MAY 22, 2024"
set_global_assignment -name LAST_QUARTUS_VERSION "13.0 SP1"
set_global_assignment -name PROJECT_OUTPUT_DIRECTORY output_files
set_global_assignment -name MIN_CORE_JUNCTION_TEMP 0
set_global_assignment -name MAX_CORE_JUNCTION_TEMP 85
set_global_assignment -name ERROR_CHECK_FREQUENCY_DIVISOR 1
```

Nous pouvons alors, et contrairement à la remarque en commentaire dans le fichier (« *Altera recommends that you do not modify this file* »), copier et réutiliser le `z80.qsf` du DE0-Nano, pour créer un `board/cycloneII/` et ajuster son contenu en remplaçant toute la première partie (9 premières lignes) par cette nouvelle configuration. La désignation des fichiers VHDL/QIP ne change pas, mais celles des broches utilisées, oui. Dans le cas de la carte en ma possession, `clk` devient `PIN_17`, `key[0]` est `PIN_144` (le bouton sur la carte) et `led[0]` à `led[7]` sont branchées sur `PIN_3`, `PIN_7`, `PIN_9`, `PIN_25`, `PIN_24`, `PIN_32`, `PIN_31` et `PIN_44` (les trois premières correspondent aux LED de la carte, mais l'ensemble sera déporté sur une platine à essais via un octuple *latch* 74HCT573 servant de *buffer* (je n'avais pas de 541 sous la main) et des résistances).

De manière générale, quelle que soit la carte Cyclone utilisée, référez-vous au schéma obtenu du vendeur (souvent, un simple message suffit pour obtenir l'URL d'un ZIP ou d'un RAR) et/ou des projets de démonstration incluant généralement un ou des fichiers QSF très instructifs. En l'absence de schéma et si le libellé des broches ne correspond pas à celui du FPGA, vous n'avez que deux options : le multimètre en mode continuité couplé de très bons yeux et des doigts agiles, ou, après avoir repéré l'horloge, utiliser un code VHDL/Verilog de type *blink* (comme le `counter` de nullobject) pour changer l'état d'une broche arbitraire et la retrouver avec une résistance et une LED. Dans les deux cas, les maîtres mots seront « patience » et « minutie » et, pour l'avoir vécu avec la seconde carte Cyclone IV (EP4CE6E22C8) dont la SDRAM ne fonctionne toujours pas, c'est vraiment quelque chose que je ne vous souhaite pas...

3.3 Ajoutons l'UART et davantage de mémoire

Nous avons bien fait le ménage et, au final, « porté » le projet initial sur pas moins de quatre FPGA Cyclone (EP2C5T144C8, EP4CE6E22C8, EP4CE6F17C8 et EP4CE15F23C8) sans trop de difficulté. Il est grand temps de passer à l'étape suivante, à commencer par ajouter l'UART. L'*IP Core* généralement utilisé dans bon nombre de projets est *gh_uart_16550* **[11]** de H. LeFevre, mais ceci est une implémentation quasi conforme du composant 16550 de National Semiconductor.

Bien que ce soit effectivement l'UART que nous ayons utilisé pour le Z80 sur platine à essais, rien ne nous oblige à reproduire toute la complexité du monde « physique » dans un FPGA. Le 16550, par exemple, peut être configuré logiciellement de bien des façons, pour s'adapter aux besoins du programmeur (vitesse, format de données, mode d'interruption, etc.). C'est un composant destiné à être adaptable et à trouver place dans nombre de réalisations physiques...

Nous, nous avons davantage de souplesse et nous pouvons simplifier les choses avec un *IP Core* bien moins complexe. Il nous suffira d'adapter le code C/ASM de configuration (le simplifier,

Attention, en fonction de la carte utilisée, il est possible que vous rencontriez des incompatibilités selon les broches choisies, pour les LED par exemple, mais également pour les signaux (RX/TX) dont nous aurons besoin par la suite. Les outils Quartus, et le *fitter* en particulier, ne manqueront pas de vous signaler le problème et même si la masse d'informations défilant sur l'écran donne le tournis, l'erreur sera bien là. Certaines broches d'un FPGA ne peuvent pas être utilisées en sortie, d'autres en entrée, c'est comme ça. Vous n'aurez qu'à opter pour une autre broche et le problème sera réglé.

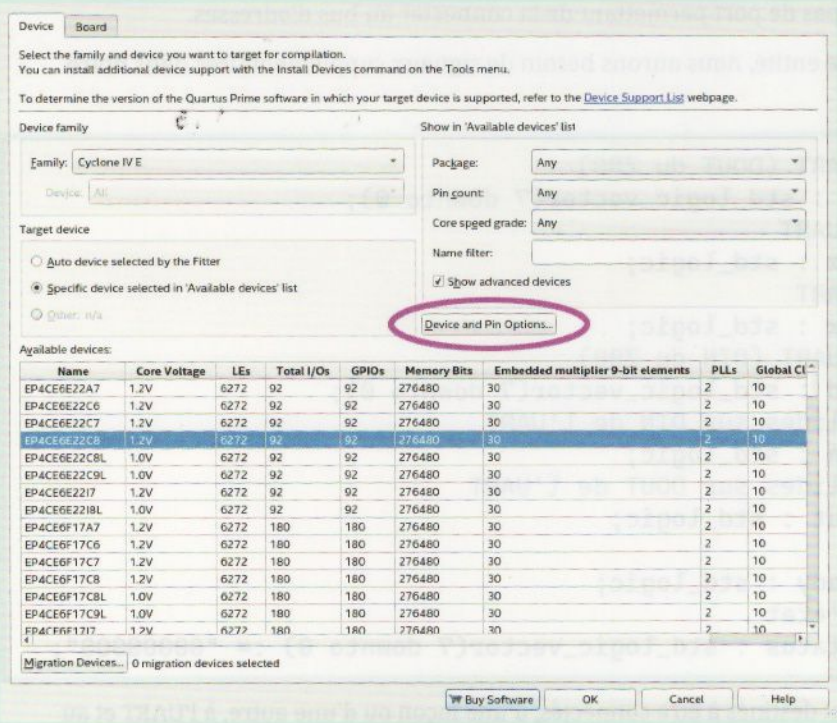
Parfois cependant, c'est le circuit de la carte lui-même qui impose le choix des broches, pour les LED qui y sont intégrées et, pourtant, vous rencontrez un problème concernant leur utilisation. C'était le cas par exemple avec ma carte Cyclone IV, sans documentation ou schéma (ce qui n'a rien arrangé à l'affaire), où la broche 101 (`PIN_101`) était physiquement connectée à l'un des segments de l'afficheur LED, mais ne pouvait pas être utilisée pour cet usage.

Le problème venait en réalité du fait que, par défaut, un certain nombre de broches ont un usage spécifique autre que des entrées/sorties standard. Il est cependant possible de changer cette configuration. Pour cela, dans l'IDE Quartus, rendez-vous dans le menu ***Assignments*** et ***Device***, ce qui aura pour effet de vous afficher la sélection du modèle de FPGA à utiliser (la même qu'à la création de projets). Là, un petit bouton ***Device and Pin Options***, noyé dans la masse, vous permet d'afficher davantage d'options parmi lesquelles vous trouverez un ***Dual-Purpose Pins*** qui vous permettra, par broches concernées, de basculer en ***Use as regular I/O***. Cliquez ***OK***, enregistrez le projet sans autre modification et une ou plusieurs lignes seront ajoutées dans le fichier QSF. Ici : `set_global_assignment -name CYCLONEII_RESERVE_NCEO_AFTER_CONFIGURATION "USE AS REGULAR IO"`. Dès lors, tout rentrera dans l'ordre et la

même) et le tour sera joué. Après maints essais donc, j'ai fini par jeter mon dévolu sur l'implémentation de Jakub Cabal, judicieusement appelée `uart-for-fpga` **[12]**, « *Simple UART for FPGA* ». Exactement ce qu'il nous faut. On récupérera le dépôt en question sur GitHub et on intégrera le contenu de son répertoire `rtl/` dans le `lib/uart/` de notre projet, sans oublier de bien référencer les nouveaux fichiers VHDL dans `z80.qsf`. Notez que Jakub fournit quelques exemples qui permettent de faciliter la mise en œuvre (`loopback`, en particulier).

Pour savoir comment instancier correctement cette entité, il nous suffit de jeter un œil au début de `uart.vhd`. On y trouve quatre paramètres de configuration :

- `CLK_FREQ`, la fréquence d'horloge utilisée pour permettre le calcul du débit, ici 50 MHz (`50e6`) ;
- `BAUD_RATE` pour le débit souhaité, non réglable logicielle-ment donc (ici `38400`) ;
- `PARITY_BIT` pour configurer la parité (`none`, `even`, `odd`, `mark` ou `space`, ici `none`) ;
- `USE_DEBOUNCER` pour activer/désactiver l'anti-rebond (ici à `True`).



fameuse `PIN_101` pourra être utilisée sans obtenir une erreur.

Dans le même ordre de petits tracas qui gâchent une après-midi, notez que certaines cartes n'utilisent pas de flash EPCS, mais un composant d'un constructeur différent. Le convertisseur intègre une liste de mémoires flash Micron (MT25QL*), Cypress/Infineon (S25FL*) et Macronix (MX25L*) utilisables dans une configuration (CDF). Cependant, point de mention de mémoires Winbond, comme la W25Q16JV qu'on retrouve assez souvent sur les cartes chinoises. Ne cherchez pas, il semblerait tout simplement que les EPCS soient tout simplement des puces Winbond, et choisir une EPCS16 par exemple fonctionnera sans problème avec une W25Q16JV...

Les ports de l'entité sont également relativement simples :

- `CLK : in std_logic` : le signal d'horloge ;
- `RST : in std_logic` : le *reset* (actif à l'état haut) ;
- `UART_TXD : out std_logic` : le signal en émission ;
- `UART_RXD : in std_logic` : le signal en réception ;
- `DIN : in std_logic_vector(7 downto 0)` : les données en entrée du composant ;
- `DIN_VLD : in std_logic` : un signal indiquant quand les données sont valides (comme un /CS) ;
- `DIN_RDY : out std_logic` : permettant de savoir si l'UART est prêt à accepter des données en entrée ;
- `DOUT : out std_logic_vector(7 downto 0)` : les données en sortie du composant ;
- `DOUT_VLD : out std_logic` : un signal indiquant si des données sont lisibles ;
- `FRAME_ERROR : out std_logic` indiquant un bit de stop incorrect ;
- `PARITY_ERROR : out std_logic` indiquant une erreur de parité.

Notez que ces trois derniers signaux en sortie ne sont valides que pour un seul cycle d'horloge et nous devrons trouver une solution pour remonter cette information au code puisque, comme vous pouvez le constater, cette entité, contrairement à un 16550, n'implémente pas de registres et n'a même pas de port permettant de la connecter au bus d'adresses.

Pour instancier cette entité, nous aurons besoin de signaux supplémentaires dans notre architecture :

```
-- DIN de l'UART (DOUT du Z80)
signal uart_d : std_logic_vector(7 downto 0);
-- Réception UART
signal uart_rx : std_logic;
-- Émission UART
signal uart_tx : std_logic;
-- DOUT de l'UART (DIN du Z80)
signal uart_rd : std_logic_vector(7 downto 0);
-- Données valides sur DIN de l'UART
signal validin : std_logic;
-- Données valides sur DOUT de l'UART
signal validout : std_logic;
-- UART prête
signal uartready : std_logic;
-- Registre d'état
signal uart_status : std_logic_vector(7 downto 0) := "00000000";
```

Tous ces signaux sont destinés à être connectés, d'une façon ou d'une autre, à l'UART et au *softcore* T80. Notez `uart_status` qui sort de nulle part, et pour cause, il est créé pour l'occasion. Nous rendrons ces huit bits, dont l'état changera en fonction de `validin`, `validout` et `uartready`, accessibles depuis le code C/ASM via les fonctions/instructions `OUT`/`IN`. Nous aurons donc, en plus des LED que nous déplaçons de l'adresse `0x00` à `0x08`, deux autres « périphériques » ou plutôt « registres » : celui d'état à `0x00` et celui pour l'envoi/réception à `0x01`.

Mais avant de nous pencher sur la manière de faire cela, et par la même occasion d'augmenter la mémoire RAM/ROM disponible au maximum de ce que le FPGA peut offrir, nous commençons par instancier notre UART ainsi :

```
uart: entity work.UART
generic map (
  CLK_FREQ      => CLK_FREQ,
  BAUD_RATE     => BAUD_RATE,
  PARITY_BIT    => PARITY_BIT,
  USE_DEBOUNCER => USE_DEBOUNCER
)
port map (
  CLK           => clk,
  RST           => reset,

  UART_TXD      => uart_tx,
  UART_RXD      => uart_rx,

  DIN           => uart_d,
  DIN_VLD       => validin,
  DIN_RDY       => uartready,

  DOUT          => uart_rd,
  DOUT_VLD      => validout,
  FRAME_ERROR   => open,
  PARITY_ERROR  => open
);
```

Notez les `open` pour les ports `FRAME_ERROR` et `PARITY_ERROR`, signifiant qu'ils ne sont connectés à rien et donc « en l'air ». Pour le reste, les signaux sont connus et nous pouvons enchaîner sur la ROM et la RAM :

```
rom : entity work.single_port_rom
generic map (
  ADDR_WIDTH => 13,
  DATA_WIDTH => 8,
  INIT_FILE  => "output_files/romcode.mif"
)
port map (
  clk  => clk,
  cs   => rom_cs and not cpu_rd_n,
  addr => unsigned(cpu_addr(12 downto 0)),
  dout => rom_dout
);

ram : entity work.single_port_ram
generic map (
```

```
    ADDR_WIDTH => 12,
    DATA_WIDTH => 8
  )
  port map (
    clk  => clk,
    cs   => ram_cs,
    addr => unsigned(cpu_addr(11 downto 0)),
    din  => ram_din,
    dout => ram_dout,
    we   => not cpu_wr_n
  );
```

Rien de très différent ici par rapport à ce qu'à initialement fait nullobject, si ce n'est `ADDR_WIDTH` qui passe à 13 bits en adaptant au passage la sélection des lignes de `cpu_addr`, et le fichier d'initialisation qui devient `output_files/romcode.mif`. Nous aurons donc 8 Kio de ROM et 4 Kio de RAM avec le Cyclone II, 32 Kio et 32 Kio sur le DE0-Nano, 16 Kio et 8 Kio avec l'EP4CE6E22C8 et enfin 32 Kio et 16 Kio avec l'EP4CE15F23C8 de la dernière carte réceptionnée. Dans tous les cas, nous sommes au maximum, sauf pour le DE0-Nano à qui il reste environ 14 % de SRAM libre, mais le Z80 ne sait adresser directement que 64 Kio de mémoire en tout. À noter qu'avec cette approche relativement simple, il n'est possible de varier la taille de la ROM et de la RAM que par exposant de 2. Avoir, par exemple, 24 Kio de ROM nécessiterait une autre approche, plus complexe pour ne pas gâcher de SRAM interne.

Il ne nous reste à ajouter que la dernière pièce au puzzle en intégrant la logique qui gouverne l'ensemble du circuit. Nous commençons par les assignations simples avec :

```
-- bit 0: données valides pour l'envoi
uart_status(0) <= validin;
-- bit 2: prêt à accepter des données à envoyer
uart_status(2) <= uartready;
-- LED en sortie
led <= led_reg;
-- RX et TX vers les broches
uartpin_tx <= uart_tx;
uart_rx <= uartpin_rx;
```

Notez que le bit 1 de `uart_status`, signalant la disponibilité de données lisibles qui viennent d'être réceptionnées, n'est pas dans la liste. En effet, comme le précise les commentaires dans le code VHDL de Jakub, ce signal (`DOUT_VLD` et donc `validout`) n'est maintenu que pendant un unique cycle d'horloge, ce qui signifie donc que nous devons mettre en place un mécanisme similaire à celui d'un vrai 16550. La logique est la suivante, si lors d'un cycle d'horloge (un front montant sur `clk`), `validout` est à `1`, nous changeons le bit 1 de `uart_status` à `1`. C'est lors de la lecture de notre registre en `0x00` que nous repasserons ce bit à `0`. En d'autres termes, une lecture des données venant d'être réceptionnées par l'UART efface le bit.

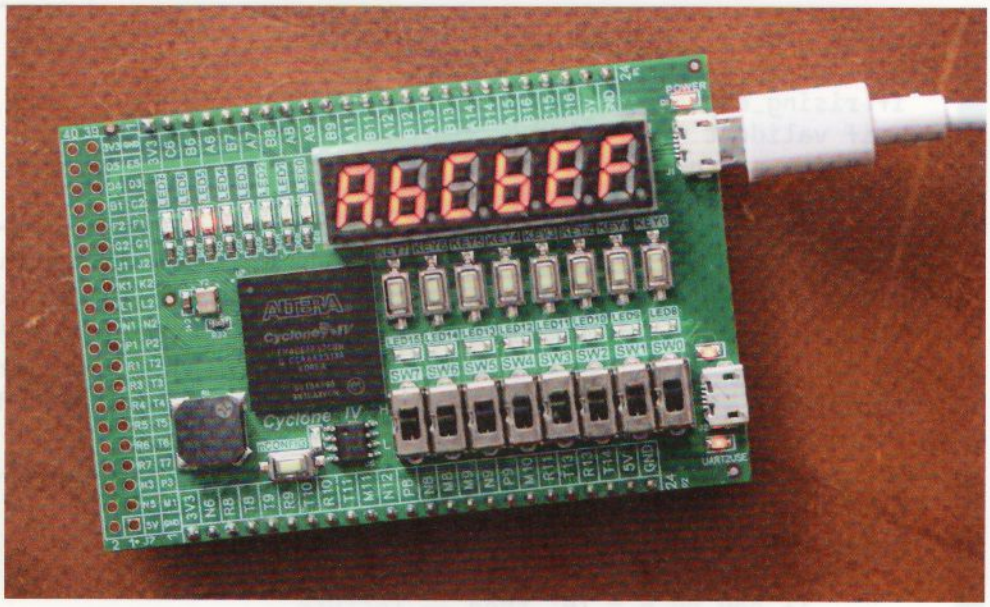
Et ceci nous amène alors précisément au gros morceau du projet : le *process* qui va gérer l'ensemble de manière synchrone, sur le front montant de l'horloge. Ce *process* débute précisément par la gestion du signal en question :

```
cpu_bus_process: process(clk)
begin
  if rising_edge(clk) then
    if validout = '1' then
      uart_status(1) <= '1';
    end if;
```

Rien de spécial ici, tout vient d'être dit. Nous pouvons passer à la gestion des accès aux périphériques et à la mémoire. Plusieurs approches sont possibles, mais j'ai choisi de simplement diviser cela entre les deux types d'accès possibles, signalés par le Z80 via ses broches /IORQ et /MREQ, respectivement « requête E/S » ou « requête mémoire ». C'est à l'intérieur de ces conditions (attention, c'est actif à l'état bas) que nous réagirons en fonction des bits du bus d'adresses et des signaux /RD (lecture) et /WR (écriture) :

```
  if cpu_ioreq_n = '0' then -- IOREQ
    case to_integer(unsigned(cpu_addr(3 downto 0))) is
      when 16#00# =>  -- accès E/S UART
        if cpu_rd_n = '0' then -- lecture
          cpu_din <= uart_rd;
          uart_status(1) <= '0'; -- clear le bit
        elsif cpu_wr_n = '0' then -- écriture
          uart_d <= cpu_dout;
          validin <= '1';  -- données valides
        end if;
      when 16#01# =>  -- accès état
        if cpu_rd_n = '0' then  -- lecture seule
          cpu_din <= uart_status;
        end if;
      when 16#08# =>  -- sortie LED
        if cpu_wr_n = '0' then  -- écriture seule
          led_reg <= cpu_dout;
        end if;
      when others =>
    end case;
  else -- si pas /IORQ
    validin <= '0';  -- données pour l'UART pas/plus valides
  end if;
```

La syntaxe `case`/`when` est similaire à un `switch`/`case` du C et nous traitons simplement adresse par adresse pour savoir avec qui le code souhaite communiquer. Une simple condition `if`/`else` sur `cpu_rd_n` et `cpu_wr_n` nous permet de faire ensuite la distinction entre lecture et écriture. C'est là, à l'adresse `0x00` et en cas de lecture que nous assignons la valeur présente sur `uart_rd` à `cpu_din`, les données en entrée du CPU et effaçons le bit 1 de `uart_status` pour accuser réception. Notez que `others` équivaut à un `default` et que celui présent ici semble ne servir à rien. En réalité, un `case`/`when` VHDL doit toujours couvrir l'ensemble des expressions possibles, donc même si ceci ne fait rien, c'est délibéré.



Voici ma dernière acquisition en date. Pour un peu plus de 30 € [17], nous avons un Cyclone IV EP4CE6F17C8 (6272 LE, 2 PLL, 180 broches E/S et 276480 bits de SRAM), 16 LED, 8 boutons, 8 interrupteurs, un buzzer, un afficheur LED 6 fois 7 segments multiplexés et... un programmeur compatible USB Blaster intégré (basé sur un MCU PIC18F14). La documentation est un peu pénible à récupérer (cloud chinois), mais le rapport ressources/prix est plus qu'acceptable.

La même structure et la même logique sont ensuite (j'ai toujours du mal à dire « ensuite », car ce n'est séquentiel que dans la description, mais pas dans l'exécution, donc c'est « ensuite en même temps ») appliquées en cas d'accès à la mémoire :

```
if cpu_mreq_n = '0' then -- MREQ
  case to_integer(unsigned(cpu_addr)) is
    -- Accès ROM 0x0000:0x1FFF 8Ko
    when 16#0000# to 16#1fff# =>
      rom_cs <= '1';
      if cpu_rd_n = '0' then
        cpu_din <= rom_dout;
      end if;
    -- Accès RAM 0x2000:0x2FFF 4Ko
    when 16#2000# to 16#2fff# =>
      ram_cs <= '1';
      if cpu_rd_n = '0' then
        cpu_din <= ram_dout;
      elsif cpu_wr_n = '0' then
        ram_din <= cpu_dout;
      end if;
    when others =>
      -- Pas vraiment nécessaire
      rom_cs <= '0';
      ram_cs <= '0';
  end case;
-- Pas un accès mémoire, pas de CS pour ROM/RAM
```

```
    else
      rom_cs <= '0';
      ram_cs <= '0';
    end if;

  end if;
end process;
```

Contrairement au C, en VHDL on peut avoir des « cas » avec une plage de valeurs. Ceci est bien pratique, car nous pouvons ainsi très facilement diviser la mémoire en différents espaces et activer le signal d'asservissement de l'une ou l'autre instance (`rom_cs` ou `ram_cs`). Pour passer d'un FPGA à un autre et en fonction de la SRAM interne disponible, c'est ici (en plus des paramètres des instances) que nous devrons ajuster les adresses des espaces mémoire. Ce code est celui du Cyclone II, je vous laisse le soin d'aller voir celui des différentes cartes Cyclone IV sur GitLab.

Ceci clôt la partie purement FPGA, mais nous devons encore toucher un mot à propos du code en C. Nous avons écarté l'option consistant à utiliser un *IP Core* 16550 et ne pouvons donc pas utiliser le code fait pour le Z80 sur platine. Cependant, les choses sont bien plus simples dans ce cas, car nous n'avons que deux registres et absolument aucune configuration (*baudrate*, format, diviseur, etc.) à faire. Notre fonction d'envoi d'un caractère sera donc :

```
int putchar(int c) {
    /* Attendre que l'UART soit prête */
    while ( !(inb(0x01) & 0x04) ) { ; }
    /* Envoyer le caractère sur les LED (port 0x08) */
    outb(0x08, c);
    /* Envoyer le caractère */
    outb(0x00, c);

    return(0);
}
```

Rien de bien extraordinaire, nous vérifions simplement le bit 4 du registre d'état puis écrivons dans le registre de données. De la même manière, la fonction de réception d'un caractère sera :

```
unsigned char getc(void) {
    /* Attendre qu'un caractère soit disponible */
    while ( !(inb(0x01) & 0x02) );
    /* Lire le caractère et le retourner */
    return inb(0x00);
}
```

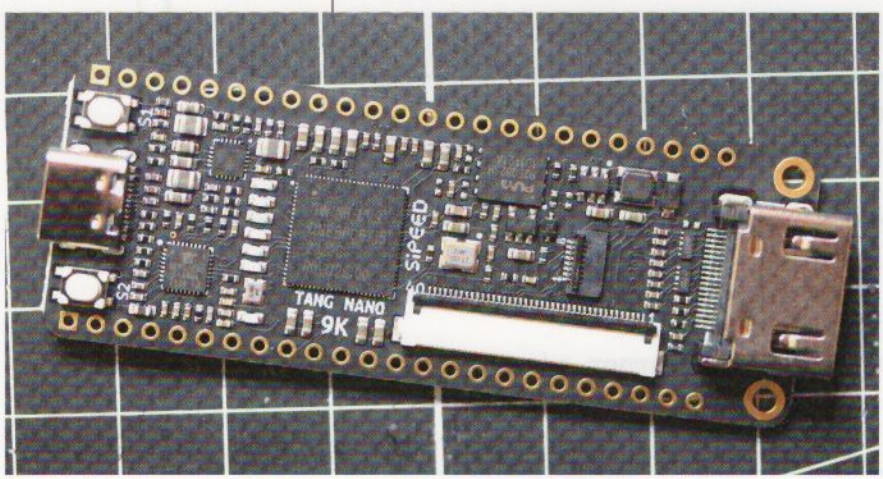
Pour profiter de tout ce travail, il suffit de compiler le code en C, convertir le binaire en MIF pour enfin synthétiser le VHDL et finalement enregistrer le *bitstream* obtenu, soit dans le FPGA, soit dans la flash EPCS. Tout ceci est automatiquement pris en charge par le `Makefile` et se fera via un simple `make load` ou `make flash`. Avec un adaptateur USB/série (3,3 volts) connecté aux broches adéquates et un émulateur de terminal (Minicom) réglé en 38400 8N1, vous devriez voir apparaître un message d'accueil après *reset* (bouton `key`) et une invitation à saisir un texte. Le texte en question, soit validé par la touche Entrée ou atteignant 31 octets, sera alors affiché à l'écran, tel que et en hexadécimal. Ceci démontrera la communication bidirectionnelle et le bon fonctionnement de la ROM, de la RAM et de la pile (et même du tas (*heap*) via un petit `malloc()` glissé dans le code).

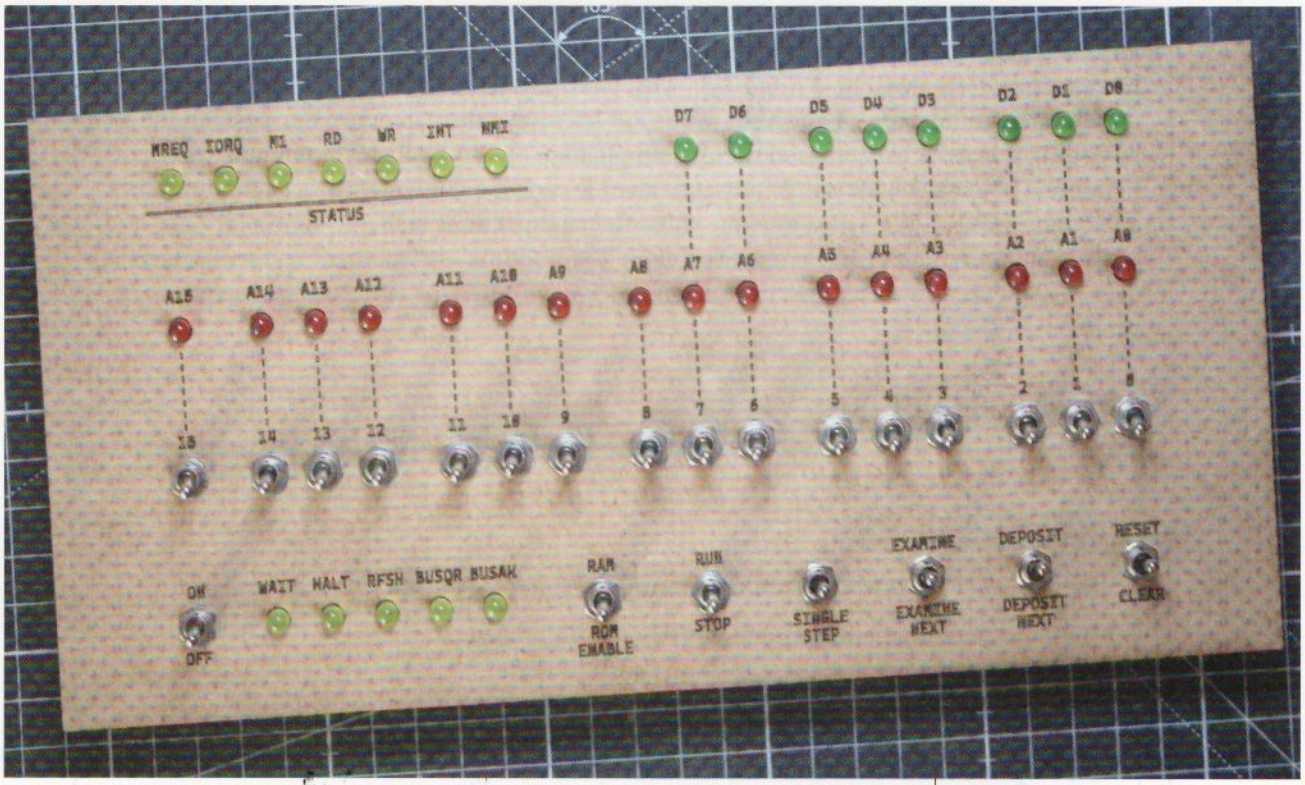
CONCLUSION

Cet article n'est clairement pas un cours de VHDL, vous l'aurez compris, mais une sorte de retour d'expérience où j'ai tenté de mettre en avant des points sur lesquels j'ai eu des difficultés à sortir de mon autoformatage de programmeur. Ceci en estimant que vous pourriez vous trouver dans la même situation et donc face aux mêmes problèmes, avec comme remède le bon vieux « c'est en faisant qu'on apprend ». Il n'est d'ailleurs pas impossible que j'aie réussi à faire hurler à la mort certains lecteurs ayant une profonde expérience du sujet et je m'en excuse. Ceci dit, Fabien, que je remercie grandement au passage, n'a pas fait d'attaque ou exprimé une quelconque envie de venir me confisquer mes cartes en voyant mon `top.vhd` et son énorme *process*, ce qui est plutôt bon signe. Il est également très probable que j'ai fait l'impasse sur un certain nombre de points que certains jugeraient importants (si vous venez de hurler « simulation » ou « timing analysis », vous avez raison), mais étant donné la taille conséquente de l'article à ce stade, je pense qu'on me pardonnera.

Cette première étape visant à avoir dans un FPGA ce qu'on avait assemblé sur platine est déjà très satisfaisante et il est maintenant parfaitement possible d'envisager d'ajouter d'autres composants ou de multiplier ceux en présence (UART). On peut

Dans la catégorie des FPGA et devkits plus récents que les « vieux » Cyclone IV (et surtout II), nous avons les cartes Tang Nano 9K (et 20K) de Sipeed. Il faudra adapter quelques éléments du projet pour en faire usage, mais pour moins de 20 €, c'est une piste qui mérite d'être explorée.





Une fois la technologie passablement prise en main, on peut commencer à envisager des réalisations plus « concrètes », comme la construction d'un ordinateur type Altair 8800 avec un joli panneau frontal plein de LED et de boutons...

ainsi prévoir des contrôleurs SPI et i2c, un composant pour piloter des WS2810b ou même imaginer gérer une sortie vidéo et un clavier, pourquoi pas. Intégrer un circuit PIO (*Programmed Input/Output*) comme le Z8420 ou le 8255A n'a que peu de sens cependant, nous avons déjà de quoi gérer une instruction `OUT`, et traiter `IN` ne sera pas très compliqué. Prévoir un ou plusieurs *timers* comme le Z8430, en revanche, peut être amusant, tout comme prendre en charge ce qui se trouve sur la carte elle-même (afficheur 7 segments tantôt).

On pourrait également envisager une autre approche pour l'utilisation des LED et de l'UART, et plutôt que d'utiliser /IORQ, *mapper* directement en mémoire les périphériques. Ceci serait un sympathique exercice, tout comme le fait de sacrifier une partie de la RAM pour créer un *framebuffer*, une mémoire vidéo utilisée, en parallèle, avec des LED adressables, une sortie VGA ou même du HDMI.

Utiliser la SDRAM présente sur certaines cartes (comme celles à base de EP4CE15F23C8 et EP4CE6E22C8) est aussi une voie d'évolution intéressante pour pallier le manque de SRAM dans le FPGA. D'autant que null-object sera, une fois encore, notre sauveur, puisque l'un de ses exemples pour DE0-Nano est précisément la lecture/écriture dans ce type de composants.

Mais en réalité, j'ai une autre idée en tête qui me paraît beaucoup plus ludique : ajouter une tripotée d'interrupteurs, boutons et LED pour obtenir quelque chose qui ressemblerait à une sorte de clone d'Altair 8800 **[13]** ou de IMSAI 8080 **[14]**. Ceci peut être réalisé en prenant comme base ce que nous avons fait ici, en ayant un interrupteur permettant de choisir si la ROM est active ou non, ou simplement en réorganisant la mémoire de manière à pouvoir/devoir entrer physiquement le code machine pour procéder à un saut à la bonne adresse. Il y a déjà quelque chose de magique dans le fait de concevoir des circuits en FPGA (ou CPLD) et ceci serait alors le summum : littéralement bouger les bits avec ses doigts et les voir s'exécuter sur un processeur qui, dans les faits, n'existe pas vraiment... DB

RÉFÉRENCES

[1] **https://connect.ed-diamond.com/auteur/marteau-fabien**

[2] **https://connect.ed-diamond.com/hackable/hk-040/chisel-construire-du-materiel-en-langage-scala**

[3] **https://github.com/nullobject/de0-nano-examples**

[4] **https://connect.ed-diamond.com/hackable/hk-044/transformez-votre-vieille-game-boy-en-console-de-salon-hdmi**

[5] **https://connect.ed-diamond.com/hackable/hk-043/execution-d-anciennes-applications-binaires-sur-un-gnu-linux-recent-l-exemple-quartus-ii**

[6] **https://savannah.nongnu.org/projects/z80asm**

[7] **https://github.com/mist-devel/T80**

[8] **https://opencores.org/**

[9] **https://gitlab.com/0xDRRB/z80vhdl**

[10] **https://github.com/trabucayre/openFPGALoader**

[11] **https://opencores.org/projects/a_vhd_16550_uart**

[12] **https://github.com/jakubcabal/uart-for-fpga**

[13] **https://fr.wikipedia.org/wiki/Altair_8800**

[14] **https://fr.wikipedia.org/wiki/IMSAI_8080**

[15] **https://www.aliexpress.com/item/1005005775201852.html**

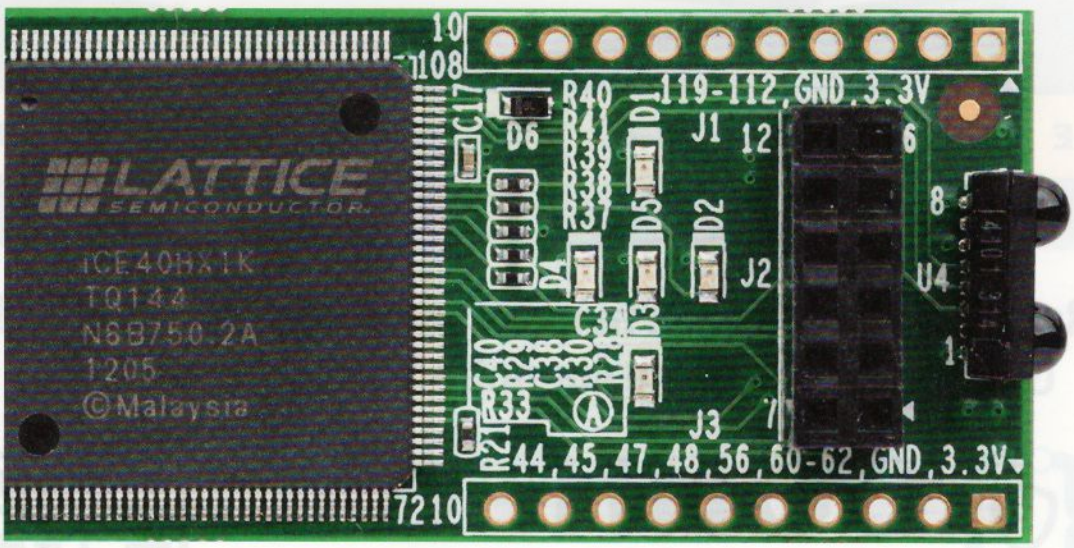
[16] **https://www.aliexpress.com/item/32812982101.html**

[17] **https://www.aliexpress.com/item/1005004691760798.html**

PIMP MY LED COUNTER, LES PERFORMANCES DE L'ADDITION

Fabien Marteau
Front de libération des FPGA

Pour évaluer un nouveau FPGA, on commence généralement avec la conception d'un compteur pour faire clignoter une LED. Ce HelloWorld simpliste nous amène à utiliser toute la chaîne de développement, de la conception du circuit en langage HDL jusqu'à la configuration du FPGA sur le kit. En passant bien sûr par la synthèse, le placement routage et le bitstream. On se penche rarement sur les performances du compteur utilisé pour le clignotement ni comment l'optimiser de manière à augmenter la fréquence de cadencement au maximum qu'il est possible d'obtenir avec le modèle étudié. C'est pourtant ce qu'on se propose de faire dans cet article à partir du kit iCEstick de chez Lattice.



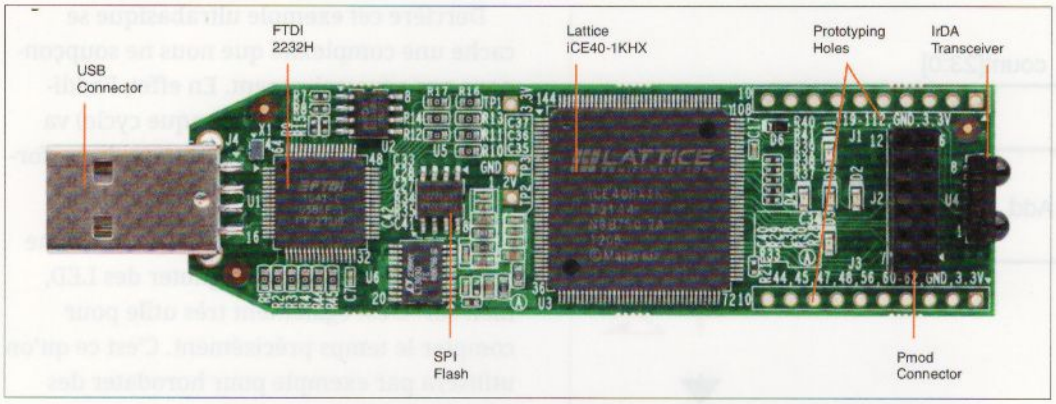


Fig. 1 : Le célèbre kit « iCEstick » de Lattice avec son horloge à 12 MHz et ses 5 LED que nous allons piloter.

Alors voilà, on vient de recevoir un nouveau kit de développement FPGA. Quelle est la première chose que nous pouvons faire avec ça ?

Faire clignoter une LED, pardi !

Prenons l'exemple du célèbre kit « clef USB » **[1]** de *Lattice*. Célèbre, car il est basé sur le premier FPGA à avoir été complètement « rétro-ingénieré » pour être utilisable 100 % avec des outils *open source* : le iCE40 (voir le feu Open Silicium **[2]**).

Le clignotement de LED est une bonne excuse pour explorer la conception de compteurs ainsi que l'opération arithmétique de base qu'est l'addition. En effet, pour être visible, la fréquence de commutation d'une LED doit descendre à quelques Hertz tandis que les FPGA sont cadencés à des fréquences allant de la dizaine de MHz jusqu'à titiller le GHz sur les modèles les plus performants. Difficile de voir une LED clignoter à l'œil avec de telles fréquences.

On peut bien sûr utiliser les *PLL* incluses dans la plupart des FPGA pour générer une fréquence basse. Mais c'est un peu du gâchis pour faire clignoter une simple LED, et surtout le code sera beaucoup moins portable d'un FPGA à l'autre, car la configuration de PLL est très spécifique à chaque modèle.

Non, le plus classique et portable pour diviser l'horloge est de compter les cycles de l'horloge qui cadence le FPGA. Si l'on veut voir la LED clignoter à 1 Hertz par exemple, la valeur maximale `c` du compteur sera proportionnelle à la fréquence de l'horloge système `Fclk` :

$$c_{max} = \frac{Fclk}{1}$$

Fig. 2

Jusqu'ici, nous avons des maths assez simples. Par exemple, l'oscillateur présent sur le kit iCEstick a une fréquence de 12 MHz, nous devrons donc compter jusqu'à 12 000 000.

Le compteur est incrémenté de 1 à chaque cycle de l'horloge principale et la LED est commutée à la moitié de la valeur max `c_max` du compteur :

- si le compteur est inférieur à `c_max/2` (6 000 000), la LED est allumée ;
- si le compteur est supérieur à `c_max/2`, la LED est éteinte.

Pour connaître les ressources occupées par le compteur, on doit calculer la taille minimum du registre binaire pour compter jusqu'à `c_max`.

$$log_2(12000000) = 23.51$$

Fig. 3

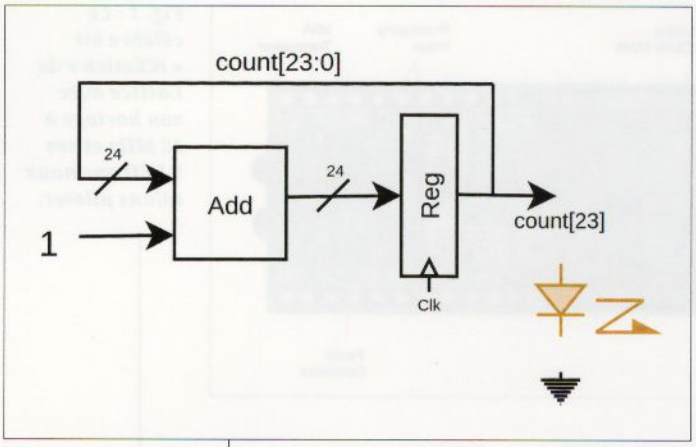


Fig. 4 : Un additionneur ajoute 1 au registre « count ». Le bit de poids fort est utilisé pour piloter la sortie branchée sur la LED.

Le résultat n'étant pas entier, il faut l'arrondir à la valeur entière supérieure (on utilise généralement la fonction `ceil` pour « plafond ») pour s'assurer d'avoir le nombre de bits nécessaire à l'écriture de la valeur maximale en binaire, soit une taille de 24 bits.

Si la précision de la fréquence de clignotement est un critère important pour notre application, nous aurons besoin de mettre en place un comparateur comme à `c_max/2`, nous venons de le voir. Ce comparateur va consommer quelques portes ainsi que du temps de calcul.

Mais si, comme dans notre cas, on veut juste voir la LED clignoter à l'œil, une fréquence qui n'est pas exactement de 1 Hz marchera également, tant qu'on la voit clignoter. Plutôt qu'un comparateur qui nécessite beaucoup de portes logiques, on pourra brancher la LED sur le bit de poids fort du registre de comptage comme on peut le voir sur la figure 4. Le bit de poids fort étant à 1 la moitié du temps, nous aurons bien le clignotement voulu avec un minimum de calcul de comparaison à faire.

Derrière cet exemple ultrabasique se cache une complexité que nous ne soupçonnons pas nécessairement. En effet, l'addition utilisée (ajout de 1 à chaque cycle) va générer des contraintes en termes de performance si le compteur est grand.

L'utilité d'un compteur dans un FPGA ne se cantonne pas à faire clignoter des LED, bien sûr. C'est également très utile pour compter le temps précisément. C'est ce qu'on utilisera par exemple pour horodater des événements (interruptions). Dans ce dernier cas, pour avoir une meilleure précision, on aura besoin d'une horloge la plus rapide possible, et donc d'un compteur rapide. Or, comme nous allons le voir, plus le compteur est grand (en nombre de bits) plus le chemin critique est long, nous obligeant à rallonger le temps de cycle en réduisant la fréquence de l'horloge.

Dans cet article, nous commencerons par faire clignoter les LED du kit iCEstick de Lattice avec un compteur « naturel ». Puis nous analyserons les performances de la méthode en changeant la taille du compteur. Nous analyserons ensuite la structure de l'addition en écrivant nous-mêmes un additionneur logique.

1. UN PEU DE DESCRIPTION MATÉRIELLE (HDL)

Tout le code décrit dans cet article est disponible sur le dépôt de l'auteur **[3]**. Les sources des exemples sont données en langage **Chisel**. C'est un langage de description matériel (HDL pour *Hardware Description Language*) basé sur **Scala** qui a été introduit dans l'article de Hackable **[4]**.

Pour diviser l'horloge, écrivons tout d'abord le compteur donné en figure 4.

La LED étant branchée sur le bit de poids fort (23) du compteur, elle s'allumera bien la moitié du temps de comptage. Avec un compteur 24 bits, la fréquence de clignotement sera de :

$$F_{blink} = \frac{(2^{24} - 1)}{12Mhz} = 1.4Hz$$

Fig. 5

On reste largement dans une fréquence de clignotement visible à l'œil.

Avec Chisel, le compteur que l'on qualifiera de naturel va ressembler à ça :

```
class NaturalCount(val COUNT_WIDTH: Int = 24) extends Module {
  val io = IO(new Bundle {
    val count = Output(UInt(COUNT_WIDTH.W))
  })

  val counterValue = RegInit(0.U(COUNT_WIDTH.W))
  counterValue := counterValue + 1.U
  io.count := counterValue
}
```

La seule entrée étant l'horloge implicite du système, nous nous contenterons d'une sortie `count` de la largeur du registre de comptage donné en paramètre du module.

Le registre `counterValue` est initialisé avec un entier non signé `UInt` de largeur demandée en paramètre (par défaut, nous prendrons les 24 bits). Le comptage s'effectue au moyen d'une simple addition avec l'opérateur `+` habituel dans tous les langages. Le retour à 0 se fera par « débordement » quand il atteindra la valeur maximale en rapport avec sa taille :

```
  counterValue := counterValue + 1.U
```

Avec ce module, on peut créer un compteur et paramétrer sa taille pour l'intégrer par la suite dans notre application.

Le module `Blink` décrit ci-dessous va instancier le compteur et s'en servir pour commuter les LED de son vecteur `leds` en sortie.

```
class Blink(val COUNT_WIDTH: Int = 24,
            val LEDS_SIZE: Int = 5) extends Module {
  val io = IO(new Bundle{
      val leds = Output(UInt(LEDS_SIZE.W))
  })

  val counter = Module(new NaturalCount(COUNT_WIDTH))
  io.leds := counter.io.count(COUNT_WIDTH - 1,
                              COUNT_WIDTH - LEDS_SIZE)
}
```

L'unique sortie du compteur est un vecteur de LED `leds` dont la taille pourra être configurée en fonction du nombre de LED disponibles sur le kit de développement. Le iCEstick ayant 5 LED (une verte et 4 rouges), nous le configurerons à 5 par défaut.

Ce module est générique, il peut être intégré sur n'importe quel FPGA. Voyons comment l'intégrer au iCE40 monté sur la carte iCEstick.

2. SYNTHÈSE DU MODULE SUR LE KIT ICESTICK

Avec le module `Blink()` présenté juste avant, on ne va pas trop se casser la tête, nous allons juste utiliser les 5 bits de poids fort du compteur pour allumer les LED. Nous aurons donc 5 clignotements aux fréquences multipliées par 2 entre chacune, la plus lente commutera à 1,4 Hz.

Pour intégrer le module `Blink()` dans l'iCEstick, nous allons devoir spécifier et brancher l'horloge et le *reset* de la classe `Module`. Avec un `Module` Chisel, l'horloge et le *reset* sont des signaux d'entrée qui sont implicites, voyons comment les connecter.

Pour les connecter, nous allons empaqueter le module `Blink` dans un module « Top » de la classe `RawModule`. Comme son nom l'indique, un `RawModule` en Chisel est un module « brut », plus d'horloge ni de *reset* implicite. Il va falloir décrire explicitement toutes les connexions.

C'est dans ce module « Top » que nous allons également préciser les macros *hardware* nécessaires au fonctionnement de notre système. En effet, si les éléments comme les LUT, les bascules *flip-flop* ou les additionneurs sont suffisamment génériques pour être inférés automatiquement par les logiciels de synthèse comme Yosys, certains éléments comme les *PLL*, *buffers* trois états et autres doivent être instanciés explicitement pour chaque modèle de FPGA. On pourrait parler de « portage » ou d'« implémentation » de notre application pour une plate-forme donnée. Quand nous voudrons porter `Blink` sur un autre FPGA, nous créerons un autre `RawModule` dans lequel nous instancierons le module `Blink`.

Le module spécifique au iCEstick ressemblera donc à ça en Chisel (`IcestickBlink.scala`) :

```
import chisel3._
import chisel3.util._

/* ❶ */
class IcestickBlink() extends RawModule {
  /* Ports d'entrée sortie ❷ */
  val clk      = IO(Input(Clock()))
  val led      = IO(Output(Bool()))
  val red_leds = IO(Output(UInt(4.W)))

  /* Domaine d'horloge "clock"
   * et de reset "false.B" (en dur) ❸ */
  withClockAndReset(clock, false.B){
    val blink = Module(new Blink(24))
```

```
    led := blink.io.leds(7)
    red_leds := blink.io.leds(6, 3)
  }
}
/* Objet compagnon correspondant à une
  fonction "main()" qui va générer le
  verilog ⓪ */
object IcestickBlink extends App {
  (new chisel3.stage.ChiselStage)
    .emitVerilog(new IcestickBlink(), args)
}
```

Le module « brut » ❶ se nomme `IcestickBlink` et est accompagné de son objet compagnon ⓪ pour générer le fichier Verilog au moyen de la commande `runMain`.

L'entrée est ici assez spécifique puisque c'est une horloge ❷. Les deux ports de sortie sont la LED verte `led` du milieu de taille 1 bit (`Bool()`) et les 4 LED rouges `red_leds` de taille 4 bits (`UInt(4.W)`).

L'instanciation du module de clignotement `Blink()` se fait dans le domaine d'horloge défini par `withClockAndReset()` avec l'horloge `clock` et un *reset* fixé « en dur » ❸.

L'encapsulation des modules que nous venons de décrire est résumée à la figure 6.

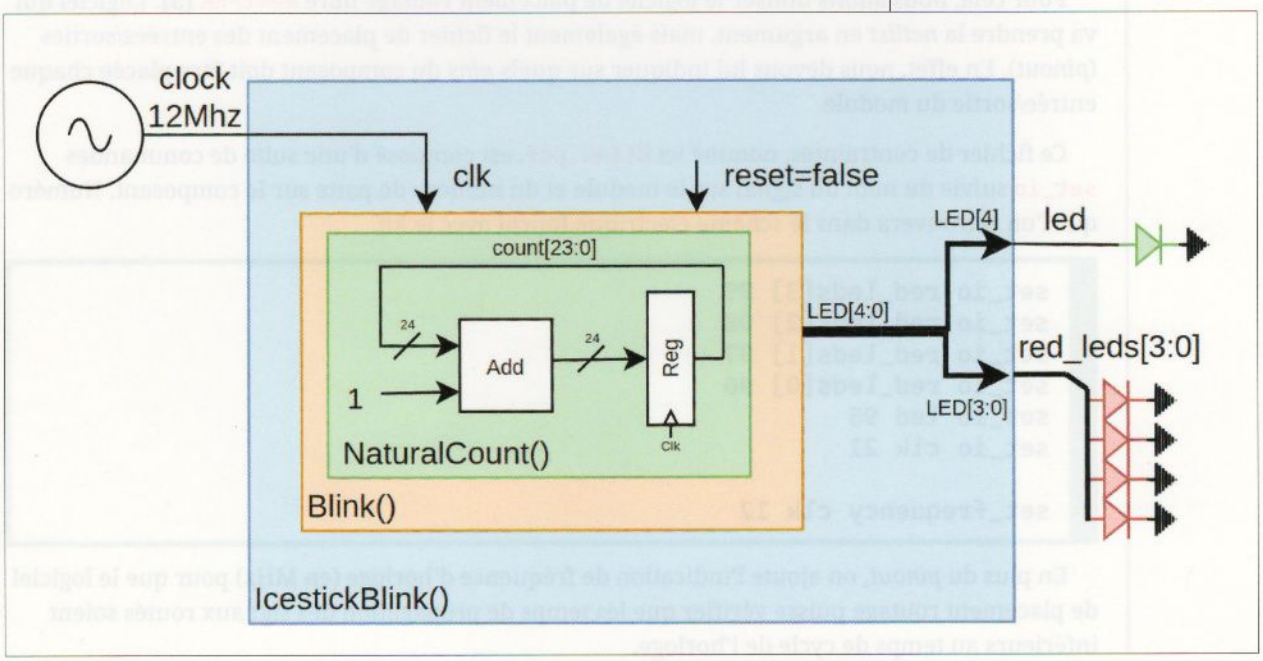


Fig. 6 : Les modules NaturalCount et Blink de ces poupées russes sont écrits en Chisel « pur ». Le module « Top » IcestickBlink est quant à lui un RawModule pour l'intégration de macros spécifiques au FPGA utilisé ainsi qu'à l'adaptation des ports entrée/sortie.

Nous en avons terminé avec la partie description matérielle (HDL), nous allons pouvoir passer à la synthèse puis au placement routage et à la configuration du FPGA.

La synthèse avec Yosys se faisant à partir du Verilog, il va falloir le générer avec la fonction `emitVerilog()`. La syntaxe `object NomClasse extends App {` définit un objet **compagnon** de la classe déclarée avant. Comme cet objet hérite de la classe `App`, il sera appelé comme une fonction `main()` avec la commande `runMain` de `sbt` :

```
$ sbt
sbt:PimpMyCounter> runMain IcestickBlink
[info] running IcestickBlink
[success] Total time: 1 s, completed 7 mars 2023 à 21:38:53
```

Il est également possible d'utiliser la commande `show discoveredMainClasses` pour avoir la liste de toutes les fonctions `main()` disponibles dans le paquet.

Quand cette commande se passe bien, on obtient un fichier en Verilog unique contenant tous les modules encapsulés dans le projet avec le nom du module *top*, ici `IcestickBlink.v`.

La synthèse consiste à transformer les sources Verilog en une **netlist** JSON au moyen de Yosys :

```
$ yosys -p 'synth_ice40 -top IcestickBlink -json IcestickBlink.json'
../../IcestickBlink.v > log_synth.txt
```

Yosys étant très verbeux, on redirigera les messages dans un fichier de *log* avec un `>`.

Maintenant que nous avons la *netlist* `IcestickBlink.json`, il faut placer et router ces éléments dans le FPGA (les connecter entre eux suivant le « schéma électronique » donné dans la *netlist*).

Pour cela, nous allons utiliser le logiciel de placement routage libre `NextPnR` **[5]**. Logiciel qui va prendre la *netlist* en argument, mais également le fichier de placement des entrées/sorties (*pinout*). En effet, nous devons lui indiquer sur quels *pins* du composant doit être placée chaque entrée/sortie du module.

Ce fichier de contraintes, nommé ici `Blink.pcf`, est composé d'une suite de commandes `set_io` suivie du nom du signal sur le module et du numéro de patte sur le composant. Numéro que l'on retrouvera dans le schéma électrique fourni avec le kit.

```
set_io red_leds[3] 99
set_io red_leds[2] 98
set_io red_leds[1] 97
set_io red_leds[0] 96
set_io led 95
set_io clk 21

set_frequency clk 12
```

En plus du *pinout*, on ajoute l'indication de fréquence d'horloge (en MHz) pour que le logiciel de placement routage puisse vérifier que les temps de propagation des signaux routés soient inférieurs au temps de cycle de l'horloge.

La commande de **NextPnR** prend les arguments suivant :

- le fichier de contraintes *pinout* (`Blink.pcf`) ;
- la matrice (`-hx1k`) ;
- le boîtier (`tq144`) ;
- le nom et le format du fichier de sortie (`IcestickBlink.asc`) ;
- ainsi qu'une option d'optimisation de *timing* (`-opt-timing`), car nous voulons avoir le meilleur chemin critique possible.

```
nextpnr-ice40 --hx1k --json IcestickBlink.json --pcf Blink.pcf --asc
IcestickBlink.asc --package tq144 --opt-timing 2> log_pnr.txt
```

Tout comme **Yosys**, **NextPnR** est très verbeux. Par conséquent, nous ferons bien attention de rediriger tous les messages dans un fichier de *log* (ici, `log_pnr.txt`).

Le fichier `IcestickBlink.asc` ainsi généré est une suite de remplissage de *tile* qui représente le contenu des blocs de RAM de configuration du FPGA. Ce fichier est au format ASCII et donc lisible avec un éditeur de texte traditionnel.

```
.comment from next-pnr
.device 1k
.io_tile 1 0
000000000000000000
000000000000000000
000000000000000000
000000000000000000
000000000000000000
000000000000000000
000100000000000000
...
```

Ce n'est évidemment pas ce format qui est reconnu par le FPGA pour le configurer. Le iCE40 ne reconnaît qu'un format binaire à l'extension `.bit`.

L'utilitaire `icepack` intégré dans le projet **icestorm** va nous permettre de générer ce *bitstream* :

```
$ icepack IcestickBlink.asc IcestickBlink.bit
```

Bitstream que nous pourrons télécharger dans le FPGA afin de le configurer avec l'utilitaire universel `openFPGALoader` :

```
$ openFPGALoader -b ice40_generic IcestickBlink.bit
Jtag frequency : requested 6.00MHz   -> real 6.00MHz
Parse file DONE
Detected: micron N25Q32 64 sectors size: 32Mb
00
```

```
Detected: micron N25Q32 64 sectors size: 32Mb
00000000 00000000 00000000 00
Erasing: [==================================================] 100.00%
Done
Writing: [==================================================] 100.00%
Done
Wait for CDONE DONE
```

`openFPGALoader` configure le FPGA et lance l'application. Le logiciel inscrit également le *bitstream* dans l'EEPROM de configuration (N25Q32). De cette manière, le FPGA se configurera automatiquement avec ce *bitstream* à chaque mise sous tension.

Si tout s'est bien passé, nous nous retrouvons avec un clignotement « lent » de la LED verte centrale et des clignotements de plus en plus rapides des LED rouges situées autour de la verte.

3. PARLONS PERFORMANCES EN RESSOURCES ET DÉLAIS (TIMINGS)

Parler de performance d'un code permettant de faire clignoter des LED peut paraître futile. C'est pourtant un excellent exemple pour introduire des sujets comme l'occupation de ressources ainsi que la fréquence maximum de l'horloge.

On trouvera toutes les informations qui nous intéressent dans le fichier de *log* que nous avons créé au placement routage avec **NextPnR** `log_pnr.txt`.

3.1 Ressources

L'utilisation des ressources du FPGA après le placement routage (NextPnR) est la suivante :

```
Info: Device utilisation:
Info:            ICESTORM_LC:    27/ 1280     2%
Info:           ICESTORM_RAM:     0/   16     0%
Info:                  SB_IO:     6/  112     5%
Info:                  SB_GB:     1/    8    12%
Info:           ICESTORM_PLL:     0/    1     0%
Info:            SB_WARMBOOT:     0/    1     0%
```

On voit que le projet utilise 27 `ICESTORM_LC` qui est la brique de base de l'iCE40 illustrée par la figure 7. Une cellule logique `ICESTORM_LC` est composée d'une table de vérité à 4 entrées (*LUT*), d'un circuit de remontée rapide de la retenue pour l'addition et d'une bascule D (*flip-flop*).

L'utilisation de 27 cellules `ICESTORM_LC` ne signifie pas qu'elles sont utilisées dans leur intégralité. Seules quelques parties peuvent avoir été utilisées à chaque fois. Pour avoir le détail, il faut regarder le rapport de synthèse de Yosys :

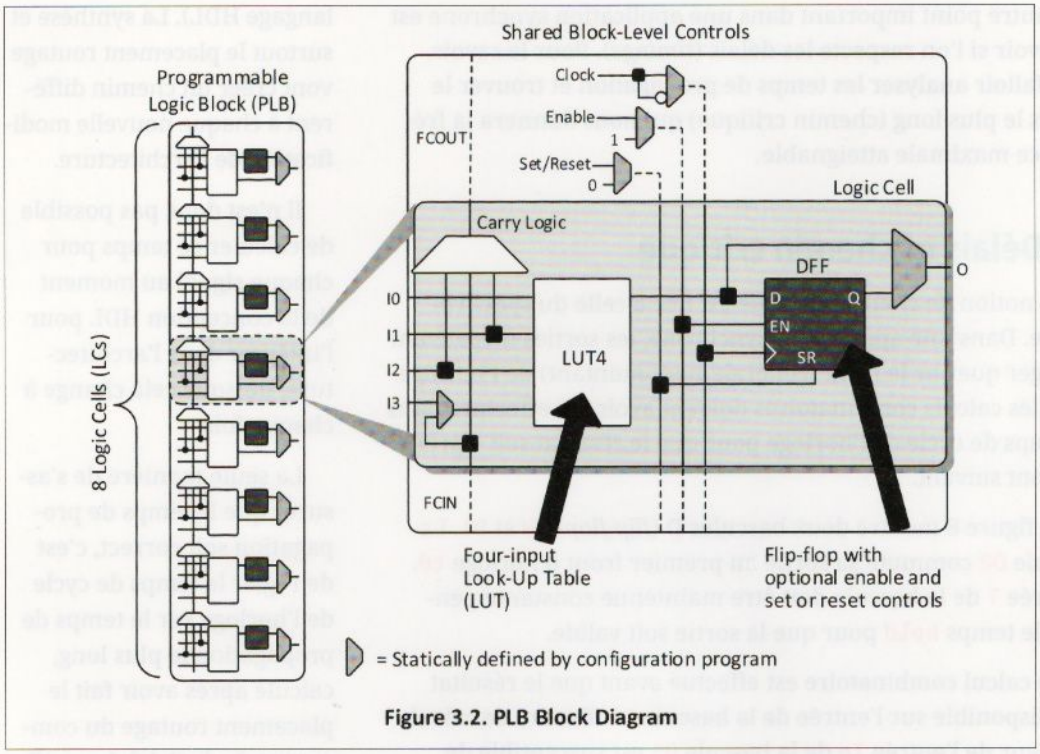


Figure 3.2. PLB Block Diagram

Fig. 7 : Schéma de la cellule logique appelée ICESTORM_LC par Yosys.

```
Number of wires:                 58
Number of wire bits:            138
Number of public wires:          58
Number of public wire bits:     138
Number of memories:               0
Number of memory bits:            0
Number of processes:              0
Number of cells:                 70
  SB_CARRY                       22
  SB_DFF                         24
  SB_LUT4                        24
```

Ce rapport nous montre qu'il y a besoin de 24 LUT à 4 entrées, 24 bascules D (DFF) et 22 multiplexeurs de retenues (CARRY). Les trois cellules `ICESTORM_LC` sont quant à elles nécessaires pour assurer le routage.

Avec 27 cellules logiques pour un compteur 24 bits, nous sommes dans l'ordre de grandeur attendu de l'occupation du FPGA. On retrouve les 5 sorties LED, l'entrée horloge dans les 6 `SB_IO` et l'unique horloge du système est branchée sur un des 8 *buffers* d'horloge du iCE40 `SB_GB`.

Le compteur rentre parfaitement dans ce petit FPGA de chez Lattice, mais qui en doutait ?

L'autre point important dans une application synchrone est de savoir si l'on respecte les délais (*timings*). Pour le savoir, il va falloir analyser les temps de propagation et trouver le temps le plus long (chemin critique) qui nous donnera la fréquence maximale atteignable.

3.2 Délais et chemin critique

La notion de chemin critique est liée à celle du synchronisme. Dans une application synchrone, les sorties ne peuvent changer que sur le front (généralement montant) de l'horloge. Tous les calculs combinatoires doivent avoir été effectués dans le temps de cycle de l'horloge pour que le résultat soit « prêt » au front suivant.

La figure 8 montre deux bascules D (*flip-flop*) `D0` et `D1`. La bascule `D0` commute sa sortie au premier front d'horloge `t0`. L'entrée `T` de la bascule doit être maintenue constante pendant le temps `hold` pour que la sortie soit valide.

Un calcul combinatoire est effectué avant que le résultat soit disponible sur l'entrée de la bascule `D1`. Durant ce calcul, la valeur de l'entrée `In` de la bascule `D1` est susceptible de prendre plusieurs valeurs différentes avant de se stabiliser à sa valeur finale.

En effet, le schéma de logique combinatoire peut inclure plus ou moins de portes logiques en cascade qui sont autant de délais qui vont modifier le temps de calcul. Ce délai est impossible à prévoir au moment de l'écriture du composant (dans un langage HDL). La synthèse et surtout le placement routage vont créer un chemin différent à chaque nouvelle modification de l'architecture.

Il n'est donc pas possible de calculer ce temps pour chaque signal au moment de la conception HDL pour l'intégrer dans l'architecture, puisque cela change à chaque fois.

La seule manière de s'assurer que le temps de propagation soit correct, c'est de régler le temps de cycle de l'horloge sur le temps de propagation le plus long, calculé après avoir fait le placement routage du composant. Le logiciel de vérification des *timings* (on parle souvent de STA pour *Static Timing Analysis*) se chargera de calculer le plus long délai une fois le placement routage effectué.

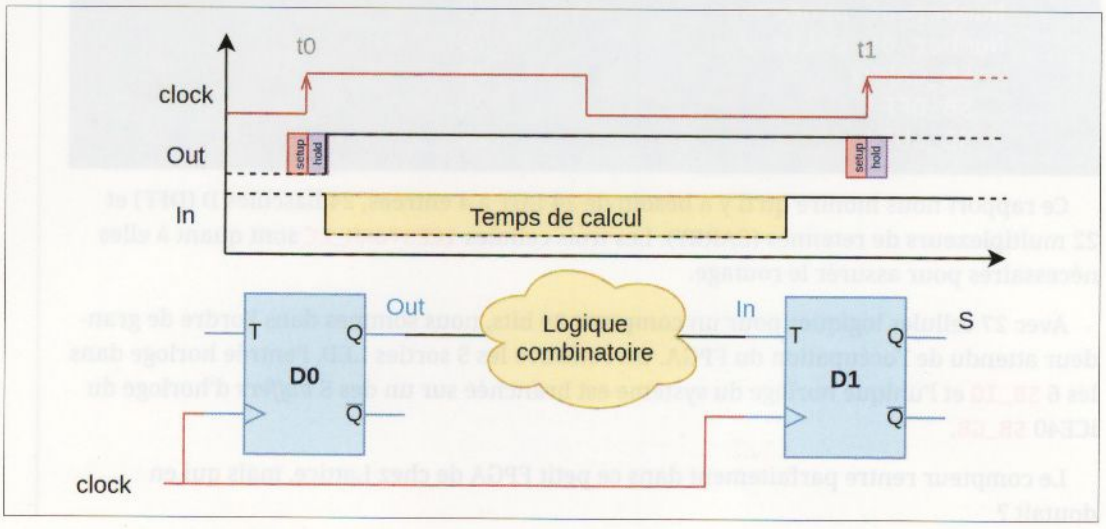


Fig. 8 : Diagramme de principe du délai de propagation.

On retrouve cette information dans le rapport de placement routage `log_pnr.txt` généré avec *NextPnR* :

• Après la synthèse :

```
Info: Max frequency for clock 'clk$SB_IO_IN_$glb_clk': 190.33 MHz
(PASS at 12.00 MHz)
```

• Après placement routage :

```
Info: Max frequency for clock 'clk$SB_IO_IN_$glb_clk': 194.33 MHz
(PASS at 12.00 MHz)
```

Ce message nous indique que les temps de propagation sont suffisants pour l'horloge de 12 MHz et que nous pourrions même monter jusqu'à 190,33 MHz avant le placement routage et 194,33 MHz ensuite.

Cela fonctionne bien, car nous avons un compteur de 24 bits, supposons maintenant que nous voulions ralentir le clignotement de la LED avec un plus gros compteur sur 32 bits.

On modifiera le fichier `icestickblink.scala` pour cela en passant de 24 à 32 l'argument de l'appel au constructeur du module `Blink()` :

```
    val blink = Module(new Blink(32))
```

Les ressources utilisées dans l'iCE40 vont être plus conséquentes, mais restent très raisonnables :

```
Info: Device utilisation:
Info:            ICESTORM_LC:    35/ 1280     2%
Info:           ICESTORM_RAM:     0/   16     0%
Info:                  SB_IO:     6/  112     5%
Info:                  SB_GB:     1/    8    12%
Info:           ICESTORM_PLL:     0/    1     0%
Info:            SB_WARMBOOT:     0/    1     0%
```

Et la fréquence maximum de l'horloge atteignable baisse à 153 MHz.

```
# Première passe
Info: Max frequency for clock 'clk$SB_IO_IN_$glb_clk': 153.56 MHz (PASS at 12.00 MHz)
# Optimisation
Info: Max frequency for clock 'clk$SB_IO_IN_$glb_clk': 157.48 MHz (PASS at 12.00 MHz)
```

Testons maintenant à 64 bits :

```
# Première passe
Info: Max frequency for clock 'clk$SB_IO_IN_$glb_clk': 86.63 MHz (PASS at 12.00 MHz)
# Optimisation
Info: Max frequency for clock 'clk$SB_IO_IN_$glb_clk': 89.56 MHz (PASS at 12.00 MHz)
```

Les performances commencent sérieusement à se dégrader, on passe en dessous des 100 MHz souvent rencontrés dans les montages FPGA. Mais nous sommes toujours bons pour la fréquence d'horloge du kit.

La question qui se pose à ce stade peut être inversée : quelle est la taille maximum de mon compteur qui rentre dans les 12 MHz de l'oscillateur ? Comme c'est à peu près sûr qu'il y aura les ressources pour compter, par dichotomie on trouve une taille de 481 bits. Avec cette taille de compteur, la fréquence de l'horloge atteignable est en dessous des 12,00 MHz à la synthèse, mais passe au-dessus après placement routage :

```
# Synthèse
Info: Max frequency for clock 'clk$SB_IO_IN_$glb_clk': 11.76 MHz (FAIL at 12.00 MHz)
# Placement routage
Info: Max frequency for clock 'clk$SB_IO_IN_$glb_clk': 12.02 MHz (PASS at 12.00 MHz)
```

Les ressources utilisées rentrent toujours parfaitement dans l'iCE40. Même si cette fois elles commencent à prendre une place non négligeable.

```
Info: Device utilisation:
Info:            ICESTORM_LC:   490/ 1280    38%
Info:           ICESTORM_RAM:     0/   16     0%
Info:                  SB_IO:     6/  112     5%
Info:                  SB_GB:     1/    8    12%
Info:           ICESTORM_PLL:     0/    1     0%
Info:            SB_WARMBOOT:     0/    1     0%
```

Mais que diantre ferions-nous d'un compteur 481 bits ? Rien. Absolument rien !

La valeur maximum de ce compteur n'a absolument aucun intérêt. Avec une horloge à 12 MHz, cela représente des temps supérieurs à l'âge de l'Univers... tout ça pour voir clignoter une LED, c'est un peu démesuré.

Ce genre de compteur ne sert pas qu'à faire clignoter des LED, c'est également utile pour compter le temps précisément. On peut s'en servir pour marquer l'arrivée précise d'un événement. Et si l'on souhaite faire la mesure sur un temps suffisamment long, il va falloir avoir une grande taille de compteur.

Et la précision du comptage va dépendre de la fréquence de l'horloge.

À 12 MHz, nous avons une précision de 83,3 ns pour compter le temps. Si l'on veut améliorer cette précision, quelle fréquence d'horloge maximum pourrions-nous atteindre avec notre compteur ?

Tout dépend bien sûr de la taille du compteur, mais pas que.

Si l'on réduit la taille à 5 bits, qui est la taille minimum pour voir les 5 LED du kit clignoter, on obtient :

```
# Synthèse puis placement routage
Info: Max frequency for clock 'clk$SB_IO_IN_$glb_clk': 423.73 MHz (PASS at 12.00 MHz)
Info: Max frequency for clock 'clk$SB_IO_IN_$glb_clk': 423.73 MHz (PASS at 12.00 MHz)
```

Avec un compteur 5 bits, on peut donc monter l'horloge du iCE40 à 423,73 MHz soit une période de 2,36 ns. Ce qui nous permet de compter le temps bien plus précisément. Par contre, à cette fréquence le clignotement des LED est invisible.

À noter au passage que la quantité de ressources utilisées est tellement faible qu'aucune optimisation n'est possible pour augmenter encore la fréquence de l'horloge au placement routage.

Nous avons ici une très bonne précision pour compter le temps. Cependant, avec un compteur de 5 bits on ne compte que jusqu'à 32 soit 32 x 2,36 ns = 75,5 ns de comptage.

Quelle précision pourrait-on obtenir si l'on voulait pouvoir compter le temps sur la base d'une journée, par exemple ? Si l'on gardait les performances du compteur 5 bits, il nous faudrait pouvoir compter jusqu'à :

$$Cmax = \frac{24 \times 60 \times 60}{2,36E^{-9}} = 36610272000000$$

Fig. 9

Soit une taille de compteur minimal de 46 bits :

$$Taille(Cmax) = ceil(log_2(36610272000000)) = 46$$

Fig. 10

Mais si l'on reporte cette taille dans l'argument du module `Blink()` de l'iCEstick, on obtient une fréquence d'horloge de 117,56 MHz :

```
# Synthèse
Info: Max frequency for clock 'clk$SB_IO_IN_$glb_clk': 113.95 MHz (PASS at 12.00 MHz)
# Placement routage
Info: Max frequency for clock 'clk$SB_IO_IN_$glb_clk': 117.56 MHz (PASS at 12.00 MHz)
```

La précision s'est donc bien améliorée puisque nous sommes à 8,50 ns. Par conséquent, la valeur max du compteur calculée précédemment nous amène à un comptage de temps supérieur à la journée. On peut augmenter la précision en diminuant la taille du compteur de manière à avoir un `Cmax` correspondant à une journée.

Par dichotomie, on trouve que pour garder une valeur maximale d'au moins une journée de comptage, on peut diminuer la taille du compteur à 44 bits.

Avec cette taille de compteur, les ressources utilisées après la synthèse Yosys sont les suivantes :

```
   Number of wires:                 98
   Number of wire bits:            238
   Number of public wires:          98
   Number of public wire bits:     238
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                130
     SB_CARRY                       42
     SB_DFF                         44
     SB_LUT4                        44
```

Les ressources utilisées après le placement routage NextPnR sont :

```
Info: Device utilisation:
Info:            ICESTORM_LC:    47/ 1280     3%
Info:           ICESTORM_RAM:     0/   16     0%
Info:                  SB_IO:     6/  112     5%
Info:                  SB_GB:     1/    8    12%
Info:           ICESTORM_PLL:     0/    1     0%
Info:            SB_WARMBOOT:     0/    1     0%
```

La fréquence d'horloge maximum obtenue est de 121,15 MHz.

```
# Synthèse
Info: Max frequency for clock 'clk$SB_IO_IN_$glb_clk': 117.32 MHz (PASS at 12.00 MHz)
# Placement routage
Info: Max frequency for clock 'clk$SB_IO_IN_$glb_clk': 121.15 MHz (PASS at 12.00 MHz)
```

Ce qui fait une période de 8,25 ns, c'est un petit peu mieux qu'avec 46 bits, mais pas énorme.

Même avec un oscillateur externe de 12 MHz, il est possible de synthétiser une fréquence supérieure grâce à la PLL présente dans la plupart des FPGA. Le détail de la configuration de cette PLL est donné dans les sources de l'article sur le dépôt de l'auteur de manière à alléger l'article.

Est-il possible d'améliorer les performances de notre compteur de manière à obtenir une période inférieure aux 8,25 ns du compteur 44 bits ? Voyons ce qu'il est possible de faire en revenant aux fondamentaux des compteurs : l'addition.

Peut-être qu'en écrivant « à la main » la logique combinatoire pour additionner, on peut grappiller quelques MHz ?

4. ADDITIONNER DANS UN FPGA

On additionne dans un FPGA de la même manière que l'on additionne avec des nombres décimaux, en ajoutant les unités avec les unités, les dizaines avec les dizaines, plus la retenue éventuelle, etc.

La différence, c'est qu'en FPGA l'addition se fait en binaire, évidemment. Donc, si les deux entrées sont à « 1 », la sortie sera « 0 » à laquelle il faut ajouter une retenue à « 1 ». Cet additionneur avec deux entrées « a » et « b » et deux sorties « c » et « s » est appelé un demi-additionneur. La table de vérité est donnée ci-dessous :

a	b	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

La valeur de sortie « s » est associée à sa retenue « c » pour *carry*. On voit ici que l'équation logique de la sortie est un « *OU* exclusif » entre les deux entrées « a » et « b » alors que l'équation de la retenue est un « *ET* logique ». Le schéma de l'additionneur est donné en figure 11.

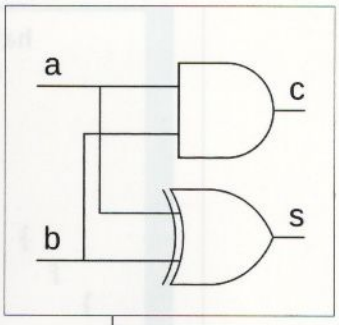


Fig. 11 : Schéma logique d'un demi-additionneur.

On appelle cet additionneur 1 bit un **demi**-additionneur, car il ne peut pas être chaîné pour obtenir une addition de nombre supérieur à 1 bit. En effet, si l'on veut le chaîner, nous aurons besoin de prendre en compte la retenue du bit de poids plus faible dans les bits d'entrée suivants. Et nous aurons besoin de trois entrées pour l'y inclure.

Le module Chisel associé au demi-additionneur décrit plus haut ressemblera au code suivant :

```
class HalfAdder extends Module {
  val io = IO(new Bundle {
      val a = Input(Bool())
      val b = Input(Bool())
      val s = Output(Bool())
      val c = Output(Bool())
  })

  io.s := io.a ^ io.b
  io.c := io.a & io.b
}
```

Pour le tester en Chisel, il faut créer un fichier source dans le répertoire `src/test/scala/` du projet et y importer le module `chiseltest`.

Nous appellerons sobrement ce code de test `BlinkTest.scala` et y ajouterons la classe de test pour le demi-additionneur :

```
import chisel3._
import chiseltest._
import org.scalatest.flatspec.AnyFlatSpec ❶

class HalfAdderTest extends AnyFlatSpec with ChiselScalatestTester {
  behavior of "HalfAdder" ❷

  it should "match the truth table" in {
    test(new HalfAdder()) { dut =>
      val half_adder_truth_table = List(
      /* a,b,c,s */
        (0,0,0,0), ❻
        (0,1,0,1),
        (1,0,0,1),
        (1,1,1,0))
```

```
        half_adder_truth_table.foreach {
          case (a, b, c, s) =>
            dut.io.a.poke(a);dut.io.b.poke(b) ❸
            dut.clock.step(1) ❺
            assert(dut.io.c.peek().litValue == c) ❹
            assert(dut.io.s.peek().litValue == s)
        }
      }
    }
  }
```

La bibliothèque Scala `AnyFlatSpec` ❶ nous donne la possibilité d'écrire les tests en anglais courant :

```
  it should "match the truth table" in {
```

Le `it` étant remplacé par « HalfAdder » avec la fonction `behavior of` ❷. Le module à tester est instancié dans `test()` et par la suite référencé par `dut`.

Pour écrire sur un port d'entrée, on utilise la méthode `poke()` ❸ et pour lire la méthode `peek()` ❹ sur le signal concerné. Enfin, on avance l'horloge par « pas » grâce à la méthode `step()` ❺ appelée sur l'horloge dont on veut avancer le temps. Ces fonctions *peek()*, *poke()* et *step()* proviennent de la classe (ou plutôt du *trait*) `ChiselScalatestTester` que l'on hérite à la déclaration du test.

Le test du demi-additionneur va consister à boucler sur une liste de *tuples* ❻ contenant les valeurs d'entrée à écrire et les valeurs de sortie attendues :

- ❸ on écrit les valeurs `a` et `b` d'entrée ;
- ❺ on avance l'horloge d'un « pas » ;
- ❹ on lit les valeurs `c` et `s` et on les compare aux valeurs attendues.

Le mot-clef `case ()` est une astuce **Scala** de *pattern matching*. Si l'élément de la liste parcourue par `foreach` est un *tuple* de 4 valeurs, il va « matcher » et remplir des valeurs les variables nommées.

Pour lancer tous les tests disponibles dans le projet, on utilisera la commande `test` dans `sbt` :

```
sbt:PimpMyCounter> test
[info] HalfAdderTest:
[info] HalfAdder
[info] - should match the truth table
[info] Run completed in 2 seconds, 496 milliseconds.
[info] Total number of tests run: 1
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 1, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
```

Ici, il n'y avait qu'un seul test dans le projet. Dans le cas où il y en a plusieurs, on utilisera plutôt la commande `testOnly` avec le nom du test si l'on veut se concentrer sur un seul test.

```
sbt:PimpMyCounter> testOnly HalfAdderTest
```

Il est également possible de lister les tests disponibles avec la commande `show` :

```
sbt:PimpMyCounter> show test:definedTests
[info] * Test HalfAdderTest : subclass(false, org.scalatest.Suite)
```

4.1 L'additionneur entier

Comme nous venons de le voir, au-delà de 1 bit, le demi-additionneur ne fonctionne plus. Nous avons besoin de reporter la retenue sur le bit de poids plus fort. Il nous faut donc une entrée supplémentaire transformant ainsi l'addition de deux nombres 1 bit en l'addition de trois nombres de taille 1 bit, comme on peut le voir sur la figure 12.

La table de vérité d'un additionneur complet est donnée ci-dessous :

a	b	cin	cout	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Nous avons cette fois deux entrées « a » et « b » auxquelles s'ajoute la retenue « cin » provenant du bit précédent et une sortie « s » accompagnée de sa retenue « cout » qui ira alimenter l'entrée de l'additionneur de bit de poids plus fort.

Si l'on considère la sortie de l'additionneur sur deux bits, avec « s » en poids faible et `cout` en poids fort, on remarque que la fonction est un compteur de bits à « 1 » sur les entrées `a`, `b` et `cin`.

On peut donc se servir du demi-additionneur précédent pour compter les « 1 » comme montré en figure 12 et construire ainsi un additionneur complet (*Full Adder*).

Le premier demi-additionneur `HA0` (HA pour *Half Adder*) est utilisé pour faire l'addition « normale » de `a` et `b`. Ses sorties sont constituées :

- d'une sortie **g**ératrice de retenue. La retenue est « générée » par les valeurs d'entrée « a » et « b » à 1 ;
- d'une sortie **p**ropagatrice de retenue. Lorsque « p » est à 1, la retenue « cin » est propagée sur « cout ».

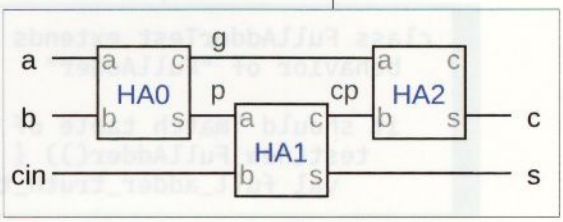


Fig. 12 : Additionneur composé de trois demi-additionneurs utilisés en compteur de « 1 ».

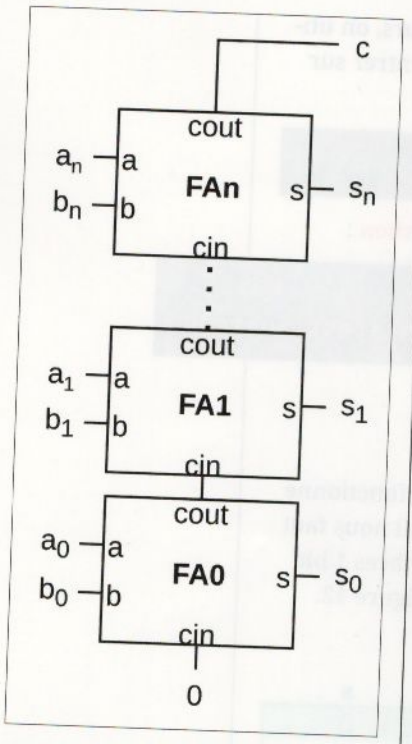


Fig. 13 : Pour additionner sur plusieurs bits, on enchaîne les additionneurs.

Comme nous n'avons que deux entrées à notre demi-additionneur, « p » et « g » ne peuvent pas être à « 1 » simultanément. Le demi-additionneur HA1 va propager la retenue « cin » si « p » est activé et la reporter sur la sortie « s » sinon.

Le dernier demi-additionneur HA2 va collecter la retenue propagée « cp » ou générée « g » pour la reporter en sortie « c ». Il est à noter que jamais « cp » et « g » ne peuvent être à « 1 » simultanément, il est donc parfaitement possible de remplacer HA2 par une simple porte logique *ou*. Et c'est d'ailleurs ce que fera le logiciel de synthèse.

Les dénominations de générateur et de propagateur sont importantes pour la conception d'additionneur rapide comme expliqué dans l'encadré.

Le module Chisel de l'additionneur complet est le suivant :

```
class FullAdder extends Module {
  val io = IO(new Bundle {
    val a    = Input(Bool())
    val b    = Input(Bool())
    val cin  = Input(Bool())
    val s    = Output(Bool())
    val cout = Output(Bool())
  })

  val ha0 = Module(HalfAdder())
  val ha1 = Module(HalfAdder())
  val ha2 = Module(HalfAdder())

  ha0.io.a := io.a
  ha0.io.b := io.b
  ha1.io.a := ha0.io.s
  ha1.io.b := io.cin
  ha2.io.a := ha0.io.c
  ha2.io.b := ha1.io.c

  io.s    := ha1.io.s
  io.cout := ha2.io.s
}
```

Pour le tester en Chisel, on peut boucler sur les 8 valeurs possibles d'entrée et vérifier la table de vérité donnée plus haut :

```
class FullAdderTest extends AnyFlatSpec with ChiselScalatestTester {
  behavior of "FullAdder"

  it should "match table of truth" in {
    test(new FullAdder()) { dut =>
      val full_adder_truth_table = List(
```

```
        /* a, b, cin, cout, s */
          (0, 0, 0,    0, 0),
          (0, 0, 1,    0, 1),
          (0, 1, 0,    0, 1),
          (0, 1, 1,    1, 0),
          (1, 0, 0,    0, 1),
          (1, 0, 1,    1, 0),
          (1, 1, 0,    1, 0),
          (1, 1, 1,    1, 1))
      full_adder_truth_table.foreach {
        case (a, b, cin, cout, s) =>
          dut.io.a.poke(a);
          dut.io.b.poke(b);
          dut.io.cin.poke(cin)
          dut.clock.step(1)
          assert(dut.io.cout.peek().litValue == cout);
          assert(dut.io.s.peek().litValue == s);
      }
    }
  }
}
```

Pour aller plus loin dans les opérations arithmétiques sur FPGA, le lecteur pourra se reporter au chapitre III du livre **[6]**.

C'est bien joli tout ça, mais pour l'instant, nous sommes toujours sur l'addition de deux nombres binaires de 1 bit. Pour compter, il va falloir élargir les opérandes à plusieurs bits.

Pour cela, nous allons chaîner des additionneurs « entiers » (*full adder*) du nombre de bits voulu, comme le montre la figure 13, ci-contre.

On comprend bien avec cette figure que la remontée de la retenue dans tous les blocs additionneurs va constituer le chemin critique pour le temps de propagation.

Le code Chisel du chaînage est donné ci-dessous :

```
/* Full Adder additionner */
class FullAdderAddition(val COUNT_WIDTH: Int = 32) extends Module {
  val io = IO(new Bundle {
    val a   = Input(UInt(COUNT_WIDTH.W))
    val b   = Input(UInt(COUNT_WIDTH.W))
    val c   = Output(Bool())
    val s   = Output(UInt(COUNT_WIDTH.W))
  })

  /* Vecteurs de connexion des entrées
   * sorties des additionneurs */
❶ val a_vec = VecInit(io.a.asBools)
  val b_vec = VecInit(io.b.asBools)
  val c_vec = Wire(Vec(COUNT_WIDTH, Bool()))
  val s_vec = Wire(Vec(COUNT_WIDTH, Bool()))
```

```
  /* Vecteur de modules additionneurs 1 bit*/
❹ val fullAdders = for(i <- 0 until COUNT_WIDTH) yield {
   val full_adder = Module(new FullAdder())
   full_adder.io.a := a_vec(i)
   full_adder.io.b := b_vec(i)
   /* Connexion des retenues de sortie sur
    * les retenues d'entrée */
❷  c_vec(i) := full_adder.io.cout
   if (i == 0) {
     full_adder.io.cin := false.B
   } else {
     full_adder.io.cin := c_vec(i-1)
   }
   s_vec(i) := full_adder.io.s
   full_adder ❺
  }

  /* Connexions des sorties */
❸ io.c := c_vec(COUNT_WIDTH-1)
  io.s := s_vec.asUInt
}
```

Pour connecter les modules additionneurs en série, nous allons utiliser des vecteurs `Vec`. En Chisel, un `Vec` est indexable pour manipuler plus facilement ses éléments.

Nous allons donc dans un premier temps ❶ transformer les deux entrées de type `UInt` en un vecteur `Vec` de valeurs booléennes. Deux vecteurs de type « fils » `Wire` sont également créés pour connecter les retenues de sortie de chaque additionneur sur les entrées retenues de l'entrée suivante ❷. La retenue de sortie est récupérée sur le dernier élément du vecteur `c_vec` et la valeur de sortie est une conversion en `UInt` du vecteur `s_vec` ❸. La génération du vecteur de module `FullAdder` est faite avec cette étrange boucle `val fullAdders = for ... yield {}` visible en ❹.

La boucle `for` va exécuter le contenu du bloc compris entre les accolades `{}` pour chaque valeur de `i` entre 0 et `COUNT_WIDTH-1`. La dernière ligne de ce bloc ❺ est une valeur de « retour » qui sera accumulée dans un vecteur par la présence du mot-clef `yield`. Ce vecteur sera stocké dans la variable `fullAdders`. Le code du bloc étant exécuté à chaque itération, on en profite pour effectuer toutes les connexions propres à chaque additionneur.

Ce code va nous permettre en peu de lignes d'avoir un additionneur de largeur configurable pour tester différentes tailles de compteurs.

4.2 Synthèse compteur basé sur un additionneur entier

Avec le compteur naturel, nous nous étions arrêtés sur un compteur 44 bits de manière à compter le temps sur une grosse journée à une centaine de MHz. Nous resterons donc sur cette taille pour l'additionneur entier, histoire de pouvoir comparer les performances.

Pour le compteur, nous remplacerons le compteur `NaturalCount` par le compteur `FullAdderCount` décrit ci-après :

```
/* FullAdder counter */
class FullAdderCount(val COUNT_WIDTH: Int = 44) extends Module {
  val io = IO(new Bundle {
    val count = Output(UInt(COUNT_WIDTH.W))
  })

  val counterValue = RegInit(0.U(COUNT_WIDTH.W))
❶ val addition = Module(new FullAdderAddition(COUNT_WIDTH))
   addition.io.a := counterValue ❷
   addition.io.b := 1.U
   counterValue := addition.io.s

  io.count := counterValue
}
```

À la place du `+` utilisé dans le compteur naturel, nous instancions le module de comptage ❶ avec en entrée `a`, ❷ le registre `counterValue` (de largeur 44 bits) et la valeur « 1 » en entrée `b`. Ainsi, à chaque coup d'horloge on ajoutera « 1 » au compteur `counterValue`, et le résultat sera accumulé dans `counterValue` pour le cycle suivant.

La synthèse et le placement routage de l'ensemble nous donnent les résultats suivants.

Ressources utilisées après la synthèse Yosys :

```
   Number of wires:                876
   Number of wire bits:           1143
   Number of public wires:         876
   Number of public wire bits:    1143
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                102
     SB_DFF                         44
     SB_LUT4                        58
```

Ressources utilisées après le placement routage NextPnR :

```
Info: Device utilisation:
Info:            ICESTORM_LC:    60/ 1280     4%
Info:           ICESTORM_RAM:     0/   16     0%
Info:                  SB_IO:     6/  112     5%
Info:                  SB_GB:     1/    8    12%
Info:           ICESTORM_PLL:     0/    1     0%
Info:            SB_WARMBOOT:     0/    1     0%
```

Avec cet additionneur, nous utilisons moins de cellules (102) que la version naturelle (130). Les cellules SB_CARRY ne sont pas utilisées, comme nous allons le voir par la suite.

```
# Synthèse
Info: Max frequency for clock 'clk$SB_IO_IN_$glb_clk': 67.91 MHz (PASS at 12.00 MHz)
# Placement routage
Info: Max frequency for clock 'clk$SB_IO_IN_$glb_clk': 65.92 MHz (PASS at 12.00 MHz)
```

Par rapport à la version « addition naturelle », les performances de notre compteur ont dégringolé. La fréquence maximum atteignable a été divisée par deux.

Le problème vient du chaînage de la retenue au travers de chaque additionneur. Une retenue surgissant dans le calcul du bit 0 doit pouvoir être propagée jusqu'au bit 44 en traversant tous les additionneurs. À chaque additionneur, un délai s'ajoute, taillant sévèrement dans les performances de calcul.

Pour accélérer le calcul, il faut trouver un moyen pour que la retenue se propage plus vite que le nombre d'étages de l'additionneur. C'est l'objet de la méthode par anticipation de la retenue *carry-lookahead* décrite dans l'encadré.

MÉTHODE PAR ANTICIPATION DE LA RETENUE CARRY-LOOKAHEAD ADDER

Pour utiliser cette méthode, nous allons reprendre notre additionneur entier de la figure 12 et lui enlever la sortie retenue « c ». Le demi-additionneur HA2 devient ainsi inutile, par contre, nous allons sortir les signaux **p**ropagation et **g**énération.

La figure 14 montre l'additionneur dépourvu de sa sortie retenue, mais muni de ses deux signaux « p » et « g ». Il est à noter que la sortie retenue « c » du demi-additionneur HA1 n'étant pas branchée, on pourrait le résumer à une porte « *OU* exclusif ». Et c'est d'ailleurs ce que fera le logiciel de synthèse.

Les deux signaux « p » et « g » seront connectés à un bloc nommé CLA pour *Carry-Lookahead* qui calculera toutes les retenues pour chaque additionneur, comme on peut le voir dans la figure 15.

La lectrice ou le lecteur qui souhaitera approfondir le fonctionnement du CLA se reportera à l'article Wikipédia **[7]**. Dans la pratique, nous ne descendrons pas si bas dans le fonctionnement d'un additionneur.

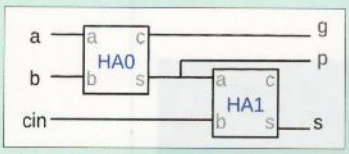


Fig. 14 : Additionneur « carry-lookahead », la sortie retenue « c » est remplacée par les deux signaux de génération « g » et propagation « p ».

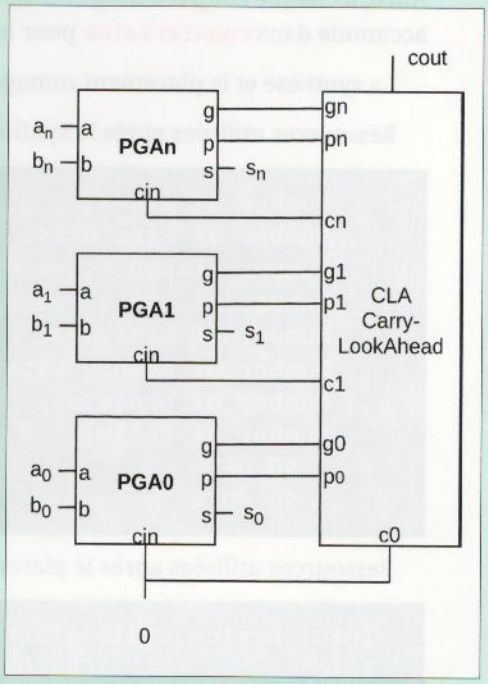


Fig. 15 : Plutôt que de chaîner chaque sortie de retenue sur l'entrée du bloc suivant, on délègue le calcul au bloc _CLA_ qui anticipera le calcul de toutes les retenues.

Il n'est pas nécessaire (ni conseillé) d'essayer de mettre en œuvre « à la main » une méthode de calcul d'anticipation de la retenue. C'est un code fastidieux, qui ne sera généralement pas bien pris en compte à la synthèse et donnera des résultats assez mauvais. Non, pour cela, nous avons un logiciel de synthèse qui sait parfaitement gérer l'addition et instancier les bonnes primitives pour optimiser au mieux. Dans le cas de l'iCE40, Yosys utilisera les macros `SB_CARRY`, comme nous avons pu le voir au début de cet article.

L'approche « conception à la main » d'un additionneur est intéressante pour comprendre son fonctionnement, mais elle ne permet pas d'améliorer les performances sur l'iCE40. Yosys et NextPnR sont parfaitement capables d'optimiser l'addition pour nous.

CONCLUSION

On ne soupçonne pas toujours jusqu'où une LED qui clignote peut nous mener. Nous avons pu nous servir de cet exemple pour analyser les performances d'un FPGA. Les performances en fréquence du compteur 44 bits sont données dans le tableau ci-après.

FPGA	Gravure	Synthèse	Placement routage	FullAdder	NaturalCount
iCE40	40 nm	Yosys	NextPnR	65,92 MHz	121,15 MHz

L'iCE40 est un FPGA avec une finesse de gravure de 40 nm. En utilisant Yosys pour la synthèse et NextPnR pour le placement routage, le rapport de performance entre la méthode « naturelle » (*NaturalCount*) et « addition à la main » (*FullAdder*) est de deux.

Nous avons vu qu'il n'est pas nécessaire de câbler un additionneur nous-mêmes, les performances de l'addition sont nettement meilleures quand on laisse les outils de synthèse et de placement routage se débrouiller tout seuls.

Les performances en fréquence des cellules logiques de l'iCE40 sont pourtant bien plus élevées que les 121 MHz trouvés. Cette fréquence de 121 MHz est-elle une limite physique du composant ? Est-il possible d'améliorer les performances de ce compteur 44 bits ?

C'est ce que nous allons voir dans le prochain article avec l'utilisation d'un compteur « rapide ». Nous testerons également les performances de deux autres FPGA : le **GateMate** et l'**EOS-S3**. **FM**

RÉFÉRENCES

[1] Clef USB à base de iCE40, **https://www.latticesemi.com/icestick**

[2] *Icestorm, une chaîne de développement libre pour FPGA*, Fabien Marteau, *Open Silicium 17*, **https://connect.ed-diamond.com/open-silicium/os-017/icestorm-une-chaine-de-developpement-libre-pour-fpga**

[3] Les sources de l'article, **https://github.com/Martoni/Diamond_HK_GLMF_OS**

[4] *Construire du matériel en langage Scala*, Fabien Marteau, *Hackable 40*, **https://connect.ed-diamond.com/hackable/hk-040/chisel-construire-du-materiel-en-langage-scala**

[5] NextPnR, logiciel libre de placement routage, **https://github.com/YosysHQ/nextpnr**

[6] *Digital Design a systems approach*, William J. Dally et R. Curtis Harting, Cambridge University Press

[7] Carry-lookhead_adder, Wikipédia (en), **https://en.wikipedia.org/wiki/Carry-lookahead_adder**

ASTERISK, RTC, PPP, CPC 464... SURFONS COMME EN 1989 !

Cédric Pellerin
Directeur technique Hoch Adler, développeur senior chez B<>Com, utilisateur de GNU/Linux depuis 1993

Quand on regarde d'anciens films ou d'anciennes séries consacrés à l'informatique – Les Petits Génies ou War Games, par exemple – la jeune génération est frappée par le fait que, pour se connecter à distance, il fallait utiliser la ligne téléphonique et un drôle de boîtier émettant des borborygmes bizarres nommé modem.

– Asterisk, RTC, PPP, CPC 464... Surfons comme en 1989 ! –

Reproduire ce mode de fonctionnement à l'heure où Orange a décidé de supprimer tout le câblage « cuivre » n'est pas totalement aisé ni même bien utile. Cependant, il reste possible de créer son propre petit Réseau Téléphonique Commuté (RTC pour les intimes) en utilisant uniquement des logiciels libres et un peu de matériel qui reste encore trouvable à des tarifs compatibles avec la morale. C'est donc l'histoire de la liaison entre un Amstrad CPC 464 et un serveur Linux que nous allons vous conter ici. Cette saga à la Lelouch se poursuivra par la mise en relation d'un de leurs descendants, un 486 sous Windows 3.11, avec Internet. Tout ceci ne sera pas franchement de tout repos, comme nous le verrons...

1. ARCHITECTURE

1.1 Cahier des charges et composants principaux

L'objectif de notre projet est donc de connecter un ordinosaure disposant d'un port série, d'un modem et d'un émulateur de terminal à un serveur Linux. Pour ce faire, nous devons recréer un RTC de toutes pièces. Le composant *open source* pour ce faire existe et se nomme Asterisk. Il s'agit d'un IPBX (de l'anglais *Internet Protocol Private Branch eXchange*, un autocommutateur téléphonique privé utilisant le protocole IP) complet capable de piloter de multiples cartes sur lesquelles on connecte soit des téléphones sur des modules nommés FXS pour *Foreign eXchange Subscriber*, soit des lignes téléphoniques analogiques sur des modules nommés FXO pour *Foreign eXchange Office*, soit un mélange des deux.

Dans le cas qui nous intéresse, nous allons juste connecter un ou plusieurs modems et un poste téléphonique analogique. Il faut donc acheter une carte munie de modules FXS. On trouve encore des cartes Digium type TDP410 d'occasion pour une trentaine d'euros sur certains sites marchands bien connus du Net. Ces cartes comportent toute la logique de connexion PCI et quatre emplacements pour des modules FXO ou FXS. Il faut donc bien vérifier de quels modules elles sont garnies avant de commander. Dans le doute, changez de fournisseur, car des modules FXO ne vous serviront à rien, à moins que vous n'ayez encore une ligne téléphonique analogique fonctionnelle chez vous et que vous vouliez prendre le risque de connecter votre serveur dessus.

1.2 Asterisk

1.2.1 Installation

Toute notre infrastructure téléphonique repose sur ce seul logiciel *open source* et largement éprouvé. Il faut faire cependant attention à plusieurs détails :

- les cartes TDP410 sont obsolètes depuis 2008 et leur *driver* n'est plus compilé par défaut dans les dernières versions ;
- Asterisk n'est pas *packagé* dans la version 12 de Debian, car le mainteneur du paquet ne s'en sort plus et il n'était pas prêt à temps ;
- Asterisk ne compile plus en 32 bits pour des raisons de divisions 64 bits.

Toutes ces raisons – qui m'ont fait perdre un temps fou – poussent à utiliser une version un peu ancienne de Linux. Pour ma part, je suis parti sur une version de Lubuntu 22.04, tout simplement parce

que je l'avais sous la main. J'ai ensuite supprimé toute la couche graphique afin de ne pas utiliser du processeur pour rien. La machine utilisée est une XW6600, station de travail HP, car elle traînait dans un coin et que je n'avais rien de mieux en architecture x86_64. Pour ceux que ça intéresse, Asterisk semble fonctionner aussi sur ARM64, mais cela fera sans doute l'objet d'un prochain article.

Le *driver* qui supporte notre carte TDM410 porte le doux nom de DAHDI, ce qui signifie *Digium Asterisk Hardware Device Interface*. Les paquets à installer sont les suivants : `asterisk`, `dahdi` et `asterisk-dahdi`.

Lors de l'installation physique de la TDM410 dans la machine, n'oubliez surtout pas de raccorder le connecteur Molex d'alimentation, sinon votre carte non seulement ne fonctionnera pas, mais elle ne sera juste pas détectée par son *driver*. Un `lspci` vous la fera voir comme contrôleur Ethernet (on ne rit pas), mais DAHDI ne la trouvera pas.

1.2.2 Configuration

La première chose à faire concerne les *drivers* DAHDI. il existe deux fichiers de configuration qui nous intéressent situés dans `/etc/dahdi` :

- `assigned-spans.conf` qui va permettre d'associer des « spans » à des numéros de canal. Dans le jargon DAHDI, un *span* est une unité logique qui peut être, par exemple, un port sur une carte numérique ou un module FXO ou FXS sur une carte analogique ;
- `system.conf` qui va permettre de numéroter les canaux et de positionner certaines variables indispensables, comme le pays dans lequel on se trouve.

Le premier fichier peut être autogénéré par la commande :

```
# dahdi_span_assignments -v auto
```

et il vaut mieux le laisser intact. Il devrait ressembler à ceci :

```
#
# Autogenerated by /usr/sbin/dahdi_span_assignments on Wed 07 Apr 2021
# 04:23:01 PM CEST
# Map devices + local spans to span + base channel number

# Device: [] @PCI_Bus_03_Slot_02
# /sys/devices/pci0000:00/0000:00:1e.0/0000:03:01.0/pci:0000:03:01.0
/sys/devices/pci0000:00/0000:00:1e.0/0000:03:01.0/pci:0000:03:01.0 1:1:1
```

Le second fichier lui sera généré en partie par la commande :

```
# dahdi_genconf -v
```

Ces deux fichiers n'ont pas vocation à être modifiés à la main, car sont susceptibles d'être écrasés par la commande ci-dessus. Cependant, pour des raisons de simplicité, nous allons les prendre comme base et les modifier selon nos besoins.

Le fichier `/etc/dahdi/system.conf` devrait ressembler à ceci :

```
# Autogenerated by /usr/sbin/dahdi_genconf on Fri Jan 26 17:58:52 2024
# If you edit this file and execute /usr/sbin/dahdi_genconf again,
# your manual changes will be LOST.
# Dahdi Configuration File
#
# This file is parsed by the Dahdi Configurator, dahdi_cfg
#
# Span 1: WCTDM/0 "Wildcard TDM410P" (MASTER)
fxoks=1
echocanceller=oslec,1
fxoks=2
echocanceller=oslec,2
fxoks=3
echocanceller=oslec,3
fxoks=4
echocanceller=oslec,4

# Global data

loadzone = fr
defaultzone = fr
```

Il faut bien penser à passer les 2 derniers paramètres à `'fr'` ou les rajouter s'ils n'y sont pas, sinon vous aurez des problèmes avec la tonalité qui ne sera pas reconnue par votre modem.

Ce script va aussi générer le fichier `/etc/asterisk/dahdi-channels.conf` qui permet de définir les numéros de téléphone associés aux canaux (donc aux *spans*) ainsi que certains paramètres supplémentaires :

```
; Autogenerated by /usr/sbin/dahdi_genconf on Fri Jan 26 17:58:52 2024
; If you edit this file and execute /usr/sbin/dahdi_genconf again,
; your manual changes will be LOST.
; Dahdi Channels Configurations (chan_dahdi.conf)
;
; This is not intended to be a complete chan_dahdi.conf. Rather, it is intended
; to be #include-d by /etc/chan_dahdi.conf that will include the global settings
;

; Span 1: WCTDM/0 "Wildcard TDM410P" (MASTER)
;;; line="1 WCTDM/0/0 FXOKS"
signalling=fxo_ks
callerid="Channel 1" <4001>
mailbox=4001
group=5
context=internal
channel => 1
```

```
callerid=
mailbox=
group=

;;; line="2 WCTDM/0/1 FXOKS"
signalling=fxo_ks
callerid="Channel 2" <4002>
mailbox=4002
group=5
context=internal
channel => 2
callerid=
mailbox=
group=

;;; line="3 WCTDM/0/2 FXOKS"
signalling=fxo_ks
callerid="Channel 3" <4003>
mailbox=4003
group=5
context=internal
channel => 3
callerid=
mailbox=
group=

;;; line="4 WCTDM/0/3 FXOKS"
signalling=fxo_ks
callerid="Channel 4" <4004>
mailbox=4004
group=5
context=internal
channel => 4
callerid=
mailbox=
group=
```

Le module `dahdi` est un module *kernel* qui est chargé « automagiquement » au démarrage de Linux grâce au fichier `/etc/modules-load.d/dahdi-linux.conf` qui contient les deux lignes suivantes :

```
dahdi
dahdi_transcode
```

Le chargement de ces deux modules provoque aussi, par effet de dépendance, le chargement des modules `wctdm24xxp` et `dahdi_voicebus`.

Cela étant, il faut dire à Asterisk de charger son propre module `dahdi`, ce qui se fait en ajoutant la ligne :

```
load => chan_dahdi.so
```

à la fin de la section `[modules]` du fichier `/etc/asterisk/modules.conf`.

Pour finir, la création des canaux DAHDI dans Asterisk s'effectue dynamiquement au démarrage et doit être initiée par la commande `dahdi_cfg`. Celle-ci semble ne plus être exécutée automatiquement lors du démarrage d'Asterisk par `systemd`, il va donc falloir mettre les mains dans la m...^W^W le cambouis et aller modifier le « script » de démarrage d'Asterisk nommé `/lib/systemd/system/asterisk.service`. Il suffit de rajouter la ligne :

```
ExecStartPre=/usr/sbin/dahdi_cfg
```

dans la section `[Service]` juste avant la ligne :

```
ExecStart=/usr/sbin/asterisk -g -f
-p -U asterisk
```

et le tour est joué.

Ceci clôt la partie configuration de la carte TDM-410 et de DAHDI. Il reste à expliquer quelques petites choses à Asterisk pour qu'il puisse fonctionner correctement.

Une première incursion dans le répertoire de configuration d'Asterisk - `/etc/asterisk` - peut se solder par des palpitations, des cheveux qui se dressent sur la tête, voire un début de panique. Inutile d'en faire trop, nous avons juste besoin de quelques petites retouches et ajouts.

Malgré les apparences, cet article n'est pas un tuto Asterisk, nous allons donc utiliser les fichiers de configuration existants, mais je ne vais pas trop m'étendre sur la signification de chaque ligne.

– Asterisk, RTC, PPP, CPC 464... Surfons comme en 1989 ! –

Tout d'abord, nous allons rajouter deux lignes d'`include` à la fin du fichier `/etc/asterisk/extensions.conf` afin de faire les choses bien et de concentrer notre configuration à nous dans des fichiers facilement identifiables. Pour ce faire, j'ai pris l'habitude – glanée quelque part sur Reddit, je crois – de préfixer les noms de mes propres fichiers par `'z_'`, ça permet de les retrouver plus facilement. Le fichier `extensions.conf` se présente donc ainsi :

```
[general]
static=yes
writeprotect=no
autofallthrough=yes
clearglobalvars=yes
priorityjumping=no

[globals]
DIAL_OPTS=g
CONSOLE=Console/dsp
MY_DIAL_STATUS=ANSWER
TIMEOUT=45

#include "z_localset.conf"
#include "z_user_context.conf"
```

Le fichier `z_localset.conf` permet d'expliquer à Asterisk ce qu'il faut faire quand on numérote sur un canal DAHDI. Ici, on reste ultra simple : « Si on compose le 4001 tu lances un appel sur le canal 1, si on compose le 4002 tu lances un appel sur le canal 2 », etc. Il se compose de quelques lignes :

```
[DAHDISets]
exten => 4001,1,Dial(DAHDI/1)
exten => 4001,2,Echo()
exten => 4001,3,Hangup()

exten => 4002,1,Dial(DAHDI/2)
exten => 4002,2,Echo()
exten => 4002,4,Hangup()

exten => 4003,1,Dial(DAHDI/3)
exten => 4003,2,Echo()
exten => 4003,3,Hangup()

exten => 4004,1,Dial(DAHDI/4)
exten => 4004,2,Echo()
exten => 4004,3,Hangup()
```

Le nom entre crochets est celui d'un « contexte », au sens Asterisk du terme. Il est possible d'en définir plusieurs. Ce contexte sera inclus dans le fichier `z_user_context.conf` qui ne comporte que deux lignes :

```
[internal]
include => DAHDISets
```

Une fois ces modifications effectuées, vous pouvez relancer Asterisk via `systemctl` et vérifier que vous avez bien une communication, en branchant par exemple deux téléphones sur la carte TDM410.

Si tout va bien, vous pouvez maintenant connecter un modem sur un canal quelconque, pour la suite nous supposerons qu'il est branché sur le 1, donc joignable via le numéro 4001.

2. CONNEXION PAR MODEM : ÉMULATION DE TERMINAL

2.1 Amstrad CPC en Basic

Un premier test de connexion possible peut se faire en utilisant un « Sasfépu » muni d'une carte RS-232 et d'un émulateur de terminal quelconque. Je vous laisse choisir votre ordinosaure préféré, pour ma part, je suis parti sur un

Eh oui, il fallait tout ça à l'époque.

Amstrad CPC 464 + lecteur de disquettes + carte série. Cette dernière est livrée avec une EPROM contenant, entre autres, un émulateur de terminal ultra simple dont il ne faudra pas attendre grand-chose, mais qui permettra de tester la communication.

À cette interface série, il va falloir connecter un autre modem relié à un autre port de la carte TDM410. Pour ma part, j'en ai profité pour tester un coupleur acoustique qui trônait sur une étagère depuis un certain temps. J'ai donc branché un téléphone type S63 sur un port FXS et le coupleur à la carte série de l'Amstrad.

Côté GNU/Linux, on va avoir besoin d'un getty prêt à répondre à nos demandes de connexion sur le modem. Étant donné que mon coupleur acoustique ne fonctionne bien qu'à 300 bauds et ne sait pas faire d'autonégociation, j'ai dû forcer la vitesse du modem « serveur » à 300 bauds en mettant la valeur 3 dans le registre S37. On en profite pour demander au modem d'être le plus discret possible (E0 : pas d'écho, Q1 : pas de code de retour) et de répondre à la deuxième sonnerie en mettant 2 dans le registre S0. Notre modem est connecté au port série géré par le *device* `ttyS0`, on communique avec lui à 9600 bauds et on demande à getty d'émuler un VT100. La commande complète devient donc :

```
# /sbin/agetty --wait-cr --init-string 'ATE0Q1S0=2S37=3\015' 9600 ttyS0 vt100
```

Ceux qui veulent pourront aller trifouiller `systemd` pour lui demander d'exécuter cette commande au *boot* et de relancer `agetty` quand il mourra.

En attendant, il est temps de tester notre communication à « longue » distance. On commence par configurer notre port série correctement sur le CPC : 300 bauds en Rx comme en Tx, 8 bits, pas de parité et surtout pas de contrôle *hardware* (RTS/CTS),

car notre modem ne le gère pas. C'est souvent le cas des modems externes comme internes, et ce, malgré ce que raconte leur configuration. Si vous n'arrivez pas à discuter avec le vôtre, pensez à désactiver le contrôle de flux *hardware* ou *software* (XON/XOFF), les problèmes viennent souvent de là.

Sur l'Amstrad, tout cela se traduit par la commande Basic :

```
|SETSIO,300,300,0,8,0,1
```

On peut ensuite lancer le terminal :

```
|TERMINAL
```

allumer le coupleur acoustique, décrocher le téléphone, composer le 4001 puis, à réception de la porteuse, poser le combiné sur le coupleur, vérifier que le voyant « Porteuse » s'allume bien fixe, et admirer les informations de *login* provenant du serveur s'afficher sur l'écran de l'Amstrad à la vitesse d'un escargot rhumatisant en plein soleil.

Youpi, ça fonctionne, on est même en 80 colonnes !

Comme signalé précédemment, le terminal est incapable de filtrer et encore moins d'émuler les codes de contrôle. On verra donc passer pas mal de caractères parasites, mais ça fonctionne. Pour corriger cela, il faut regarder du côté de terminfo et de termcap sur votre serveur. Je vous laisse faire si vous avez quelques heures à occuper.

2.2 Amstrad CPC sous CP/M

Il existe sous CP/M un émulateur de terminal assez basique nommé Kermit. Il est téléchargeable à **[CPM]** ainsi que sa documentation. Il est disponible pour une large gamme de machines sous CP/M-80 v2 ou v3, CP/M-86 et un nombre impressionnant d'OS de l'époque (VMS, Lisp Machine, Microware OS-9, etc.).

Son utilisation est assez simple. Sous CP/M 2.2 sur un Amstrad, il faut lui *setter* le *device* TTY avant de se connecter. Voilà ce que ça donne :

Fonctionne aussi sous CP/M.

Le résultat est le même qu'avec le terminal sous Basic, mais on doit pouvoir améliorer les choses en utilisant l'émulation VT52 que Kermit connaît.

3. CONNEXION PAR MODEM : ACCÈS À INTERNET

Nous voilà revenus trente ans en arrière, à une époque où l'ADSL n'existait pas encore et où la connexion à Internet se faisait à des débits qui laisseront les jeunes totalement incrédules. Cela étant, que ne ferait-on pas pour retrouver le doux bruit de crécelle du modem qui négocie sa vitesse avant de se connecter ?

3.1 Le côté serveur

L'aspect serveur peut se décomposer en trois parties :

- un *daemon* qui permet la connexion venant du port série ;
- un service ppp permettant de *tunneliser* la connexion précédente ;
- une règle de *firewalling* afin de *masquerader* tout ça.

3.1.1 mgetty

La connexion arrivant sur le port série sera prise en charge par `mgetty`. Ce dernier est en effet beaucoup plus souple que `agetty` et consorts, et il est prévu pour initier une connexion `ppp`. Cependant, j'ai été confronté encore une fois au problème de contrôle de flux du port série. Par défaut, `mgetty` utilise un contrôle *hardware*, mais mon modem, pourtant un modem PCI donc pas un problème de câble, ne le gère pas correctement. Pour changer ce « flow control », il faut impérativement recompiler `mgetty`, aucune option de configuration ne permettant de le faire ! C'est très bien indiqué à la fin de la `manpage` :

```
BUGS
       Not all of mgetty configuration can be done at run-time yet. Things
like flow control and file paths (log file / lock file) have to  be
configured
       by changing the source and recompiling.

       Users never read manuals...
```

Le plus simple sous Debian ou dérivé est d'installer les sources via `apt source` et ensuite de les adapter à nos besoins. On va commencer par le fichier `policy.h` :

- ligne 304, j'ai passé la vitesse par défaut de 38400 bauds à 19200 ;
- ligne 363, dans `#define DEFAULT_MODEMTYPE`, remplacer « `auto` » par « `data` », car on ne fera pas de fax ;
- le plus important est ligne 426, il faut remplacer `FLOW_HARD` par `FLOW_NONE` dans `#define DATA_FLOW`.

Les numéros de ligne correspondent à mgetty 1.2.1, ils peuvent légèrement varier en fonction de votre version.

Pour simplifier, j'ai aussi remplacé :

```
LIBDIR=$(prefix)/lib/mgetty+sendfax par LIBDIR=$(prefix)/lib/mgetty
```

et :

```
CONFDIR=/etc/mgetty+sendfax par CONFDIR=/etc/mgetty
```

dans le `Makefile`.

Une fois ces modifications effectuées, les classiques `make` et `make install` suffiront à compiler et installer tout ça.

Si vous tombez sur une erreur de compilation concernant « `sys_nerr` » et/ou « `sys_errlist` », c'est que votre glibc, comme la mienne, n'est pas assez antique. Pour régler le problème il faut appliquer le patch suivant dans `logfile.c` ligne 333 à 336.

Remplacer :

```
log_infix, ws,
     ( errnr <= sys_nerr ) ? sys_errlist[errnr]:
     "<error not in list>" );
#ifdef SYSLOG
```

par :

```
     log_infix, ws,
     strerror(errnr));
#ifdef SYSLOG
```

D'après le GitHub, `mgetty` n'a pas été retouché depuis 12 ans ! Si quelqu'un veut reprendre, ça rendrait un fier service à la communauté ordinosaure...

La configuration de `mgetty` commence par la création d'un fichier de démarrage pour `systemd` que nous allons nommer `/lib/systemd/system/mgetty.service` :

```
[Unit]
Description=External Modem
Documentation=man:mgetty(8)
Requires=systemd-udev-settle.service
After=systemd-udev-settle.service

[Service]
Type=simple
ExecStart=/sbin/mgetty /dev/ttyS0
Restart=always
PIDFile=/var/run/mgetty.pid.ttyS0

[Install]
WantedBy=multi-user.target
```

Côté `mgetty`, le seul fichier à modifier est `/etc/mgetty/mgetty.config`. Le plus simple est de renommer l'original et de repartir à zéro avec une configuration simple :

```
debug 5

port ttyS0
 port-owner root
 port-group dialout
```

```
port-mode 0660
data-only yes
ignore-carrier no
toggle-dtr yes
toggle-dtr-waittime 500
rings 1
#autobauding yes
speed 19200
```

Inutile de détailler chaque ligne, tout est plutôt clair. En ce qui concerne le *debug*, il peut monter à 9, mais à 5 il est déjà suffisamment verbeux pour comprendre ce qui ne va pas.

Maintenant, nous pouvons valider le service `mgetty` via la commande :

```
# systemctl enable mgetty.service
```

puis le démarrer comme d'habitude :

```
# systemctl start mgetty
```

On peut déjà valider en appelant le 4001 avec un téléphone et vérifier que le modem décroche bien à la première sonnerie.

3.1.2 pppd

Le service pppd n'est guère plus compliqué à configurer. On va commencer par créer un utilisateur nommé « `dial` » (par exemple) chargé de le lancer au *login* :

```
# useradd -G dialout,dip,users -m -g users -s /usr/sbin/pppd dial
```

et on lui affecte un mot de passe.

Cet utilisateur appartient aux groupes « qui vont bien » pour avoir les droits d'accès au port série et son shell est `/usr/sbin/pppd`, ce qui permet de lancer ce dernier à la connexion.

Maintenant, on peut rajouter cet utilisateur et son mot de passe, en clair, à la fin du fichier `/etc/ppp/pap-secrets`. On peut trouver cette façon de faire bien peu sécurisée et c'est parfaitement exact, mais à l'époque on ne se posait pas trop ce genre de question. Accéder au fichier `pap-secrets` nécessitait un accès `root` sur le serveur, ce qui paraissait suffisant comme garantie. *O tempora, o mores* !

Il faut penser à mettre le mot de passe entre guillemets, ce qui donne :

```
dial * "toto" *
```

On peut maintenant s'attaquer au fichier d'options de pppd, le bien nommé /etc/ppp/options. Là encore, on repart « from scratch », c'est nettement plus simple :

```
# Define the DNS server for the client to use
ms-dns 1.1.1.1
# async character map should be 0
asyncmap 0
# Require authentication
auth
# We want exclusive access to the modem device
lock
# Show pap passwords in log files to help with debugging
show-password
# Require the client to authenticate with pap
+pap
# If you are having trouble with auth enable debugging
debug
# Heartbeat for control messages, used to determine if the client connection
# has dropped
lcp-echo-interval 30
lcp-echo-failure 4
# Cache the client mac address in the arp system table
proxyarp
# Disable the IPXCP and IPX protocols.
noipx
```

Cette configuration est suffisante, mais elle reste améliorable, bien entendu.

Il s'agit là des options générales, il reste à préciser les options pour chacun des ports série connectés à un modem. Ici, c'est simple, on n'en a qu'un. Pour ce faire, on crée un fichier nommé /etc/ppp/options.ttyS0 pour le port /dev/ttyS0 et on y met les options spécifiques :

```
local
lock
nocrtscts
noipdefault
192.168.32.1:192.168.32.10
ipcp-accept-local
ipcp-accept-remote
netmask 255.255.255.0
noauth
proxyarp
defaultroute
lcp-echo-failure 60
lcp-max-configure 45
idle 600
```

C'est ici que l'on spécifie les adresses IP côté serveur (192.168.32.1) et celle du client qui se connectera sur ce port (192.168.32.10). Il s'agit là d'adressage fixe, d'autres modalités sont possibles, ça vous fera un excellent exercice...

3.1.3 Forward de paquets et firewall

Pour terminer, il faut expliquer à notre serveur quoi faire des paquets arrivant sur son port série. Il faut les *forwarder*, cela se fait comme d'habitude en mettant la ligne :

```
net.ipv4.ip_forward=1
```

dans `/etc/sysctl.conf`.

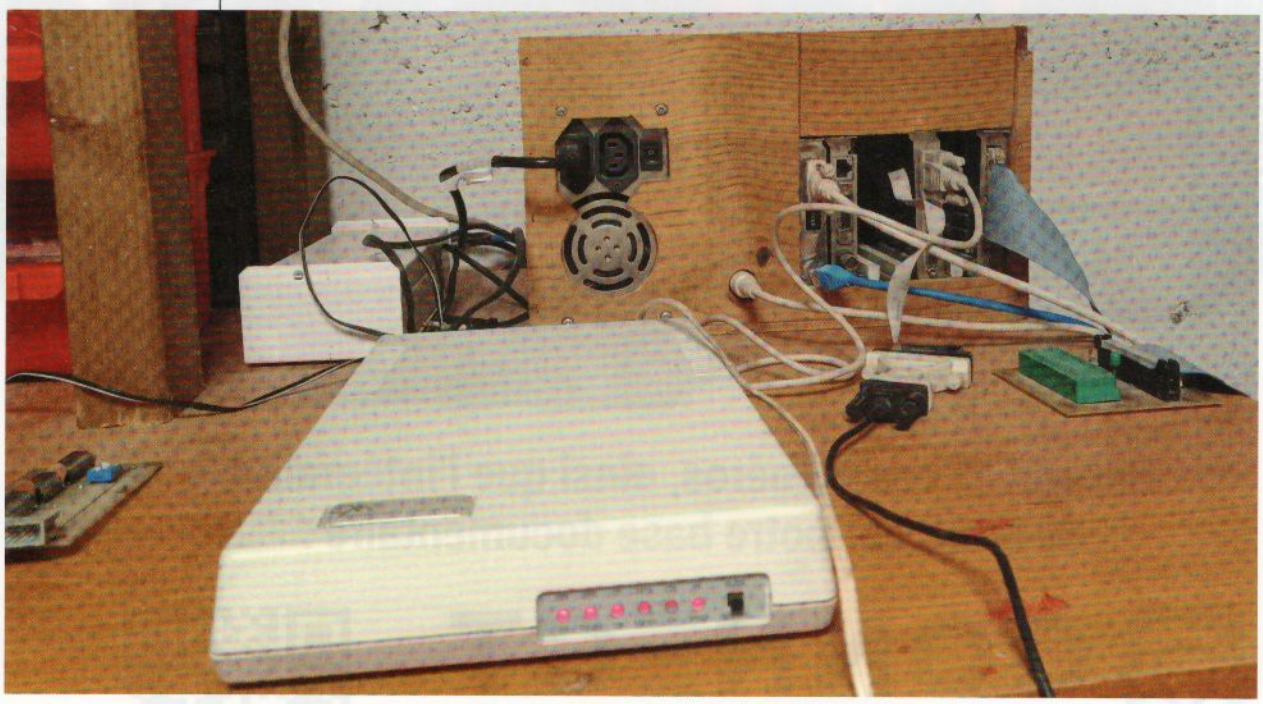
Enfin, il faut demander au *kernel* de *masquerader* tout ce qui vient du tunnel ppp :

```
# iptables -t nat -A POSTROUTING -s 192.168.32.0/24 -o enp14s0 -j MASQUERADE
```

Ici, `enp14s0` est le doux nom de ma carte réseau, à adapter en fonction de votre configuration bien entendu. Si vous en restez là, la configuration netfilter n'est pas pérenne et vous devrez retaper cette commande après chaque *reboot*. Vous pouvez jouer avec `iptables-save` et `iptables-restore` ou utiliser `systemd` pour lancer la commande à chaque *reboot*. Je persiste à dire que c'était franchement plus simple du temps de sysv init, de `/etc/inittab` et de `/etc/rc.local`...

Le serveur ppp n'étant pas un *daemon*, inutile de chercher à relancer quoi que ce soit après chaque modification.

Comment monter un 486 quand on n'a pas de boîtier.



3.2 Client Windows 3.11

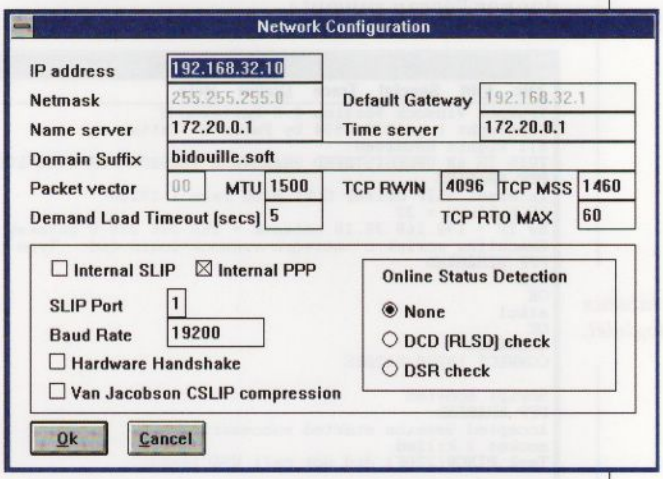
Pour tester notre configuration, nous allons faire un bond d'environ dix ans dans le temps depuis notre Amstrad et utiliser un 486 sous Windows 3.11. En effet, même s'il existe des *stacks* TCP/IP pour Z80, je ne connais pas de navigateur web pour CPC.

Avant de pouvoir nous connecter, il nous faut regrouper les éléments suivants :

- un compatible PC à base de 486 avec pas mal de mémoire, genre 8 ou 16 Mo ;
- un disque dur sur lequel on a installé un MS-DOS 6.22 ou équivalent ainsi qu'un Windows for Workgroup 3.11. Cette version est la moins *buguée* de l'ensemble, c'est pourquoi je la recommande plutôt qu'un Windows 3.1 ;
- un modem externe ou interne, au choix (voir figure ci-contre) ;
- le logiciel Trumpet Winsock en version 2.0, car la 3.0 manque d'options dans la configuration réseau et je n'ai jamais réussi à la faire fonctionner ;
- Netscape 1.6, histoire d'être bien dans l'ambiance...

Commençons par installer et configurer Trumpet Winsock. Ce logiciel s'installe en copiant les fichiers dans un répertoire, par exemple `C:\WINSOCK` et en rajoutant ce dernier dans le PATH (éditer `C:\AUTOEXEC.BAT`, modifier la ligne `PATH` et *rebooter*). Ensuite pour l'exécuter, il faut ouvrir le navigateur de fichiers et aller double-cliquer sur `TCPMAN.EXE` dans `C:\WINSOCK`.

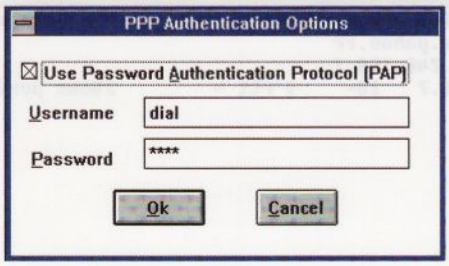
L'écran de configuration s'obtient via le menu ***File → Setup*** :



La configuration réseau.

L'adresse IP à renseigner est celle définie pour le client dans le fichier `options.ttyS0` du serveur. Les autres informations sont classiques et dépendent de votre LAN. L'adresse de la *gateway* est, bien entendu, celle du serveur faisant rouler Asterisk, dans la plage d'adresses du tunnel PPP.

Il reste à renseigner les informations de *login* PAP dans ***File → PPP Options*** :



Login et password définis sur le serveur.

Pour finir, il va falloir retoucher le script `LOGIN.CMD`, car `mgetty` ne va pas présenter la bannière attendue par défaut. Il faut remplacer, en ligne 22 :

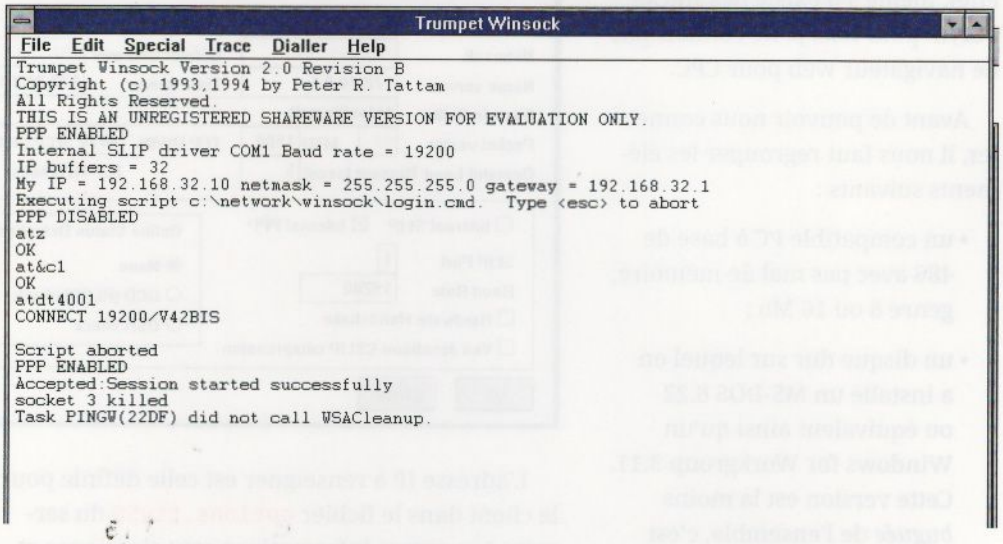
```
$userprompt = "sername:"
```

par :

```
$userprompt = "ogin:"
```

Une fois ce fichier sauvegardé, il est temps de tenter une connexion en lançant `TCPMAN.EXE` puis dans le menu ***Dialler***, sélectionner ***Login***. Cette action devrait nous donner l'écran suivant :

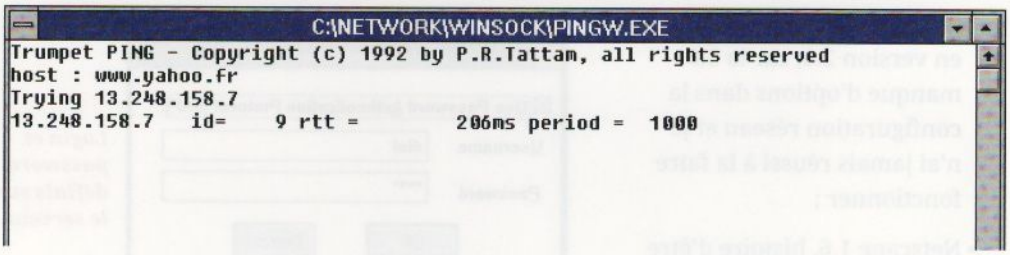
Un peu taiseux comme logiciel.



sur lequel l'information importante est : « *Session started successfully* ».

Une fois cette étape primordiale passée, on peut vérifier rapidement la bonne connexion en utilisant le logiciel `PINGW.EXE` sis dans `C:\WINSOCK`. On devrait avoir un résultat semblable à celui-ci :

Ping de Yahoo à 206 ms !



Le temps de *ping* prouve bien que l'on passe par le modem...

Enfin, nous pouvons lancer Netscape et naviguer le grand Ternet, enfin presque. Oui, il faut trouver des sites suffisamment anciens pour parler HTML 1.0 et pas HTML 1.1 ou supérieur. En effet, 99 % des sites actuels ajoutent des informations de `charset` après le `mimetype text/html` que Netscape 1.6 ne comprend pas. J'ai trouvé le site d'un passionné de radio nommé **http://www.k7tty.com** qui respecte la norme. Il est rempli de photos et à 14400 bauds, on a largement le temps d'aller déjeuner, mais

il permet de valider notre montage. Le site **http://mosaic.mcom.com** passe aussi, de même que **http://frogfind.com**, comme on pourrait s'en douter...

Une fois suffisamment fait joujou sur le Net des années 90, pour se déconnecter, il faut choisir le menu ***Dialler → Bye*** dans Trumpet Winsock, écraser une larmichette de nostalgie et revenir aux affaires courantes.

4. TROUBLESHOOTING

En cas de souci, voici quelques réflexes à avoir quand on utilise des liaisons RS232 :

1) Vérifier que le câble est le bon. En effet, il existe deux types principaux de câbles série : le câble droit (Rx → Rx, Tx → Tx, RTS → RTS, CTS → CTS, etc.) et le câble croisé (Rx → Tx, Tx → Rx, RTS → CTS, CTS → RTS, etc.).

   En règle générale, un câble droit est utilisé pour connecter un *Data Terminal Equipment* (DTE) à un *Data Communication Equipment* (DCE). Le premier se réfère à un ordinateur ou un terminal passif, tandis que le deuxième est principalement un modem. Le câble croisé est utile lorsque l'on veut connecter entre eux directement deux ordinateurs ou brancher un terminal sur un ordinateur. Attention donc de ne pas les confondre.

2) Ne jamais hésiter à éteindre et rallumer un modem lorsqu'il ne répond plus ou semble faire « n'importe quoi ». Ces équipements sont assez sensibles et parfois prompts à planter au pire moment. C'est pourquoi il est recommandé de privilégier des modems externes qui peuvent être manipulés sans avoir à *rebooter* l'ordinateur.

Ça rappelle des souvenirs.

3) Je le répète une dernière fois, attention au contrôle de flux. Commencez par le désactiver et, une fois que tout ira bien, vous pourrez tenter de le remettre en fonction.

4) Un modem « récent » parle quasi systématiquement Hayes. Il s'agit d'un langage de contrôle dédié aux modems qui date de 1981 et qui s'est enrichi petit à petit jusqu'à maintenant. Il est toujours utilisé pour contrôler les modems 4G et 5G entre autres. Du coup, minicom ou microcom sont vos amis. N'hésitez pas à les utiliser pour vérifier que votre modem répond bien, le *resetter* (ATZ) et essayer de numéroter à la main (ATDT<numéro>). La liste des commandes est longue comme plusieurs bras, un PDF plutôt bien fait mais consistant (614 pages) est accessible ici : **[HAYES]**.

5) L'utilisation de matériel ancien est toujours délicate, on n'est jamais à l'abri d'un faux contact, de connecteurs oxydés ou sales, etc. Les bombes de nettoyant pour contacts peuvent vous sauver la vie.

6) Pour les plus bricoleurs d'entre vous, il est assez facile de construire un *sniffeur* de connexion RS232 en repiquant sur une des prises du câble les signaux Rx, Tx et GND. Vous connectez ensuite cette dérivation sur un terminal ou sur un *laptop* et vous pourrez regarder passer les octets. Un schéma est disponible ici : **[SNIF]**.

CONCLUSION

Ce petit article est initialement le fruit d'une réflexion entendue lors d'une présentation de « Sasfépus » dans mon village. Un visiteur a juste dit qu'il était dommage de ne pas pouvoir montrer aux jeunes la façon dont on se connectait à Internet dans les années 90 avec un modem. Cette phrase a fait tilt dans ma tête et je me suis mis à monter cette plateforme pour pouvoir mettre à disposition un accès à Internet par modem lors de la prochaine exposition. J'aimerais juste pouvoir remplacer la XW6600 (140 watts et 10 kg) qui me sert de serveur actuellement par une Raspberry Pi ou équivalent. Il existe des *shields* FXS pour RPi, mais difficiles à trouver. Si quelqu'un a envie qu'on s'y mette à deux, je suis preneur :).

Dans l'intervalle, vous avez tout ce qu'il faut pour montrer à la jeune génération qu'on pouvait parfaitement faire de l'informatique avec les moyens de l'époque et que, au moins, on maîtrisait nettement mieux notre environnement. Oui, on peut développer avec 64 Ko de mémoire, oui on peut accéder au Net à 14400 bauds et, en plus, on était bien content de pouvoir le faire.

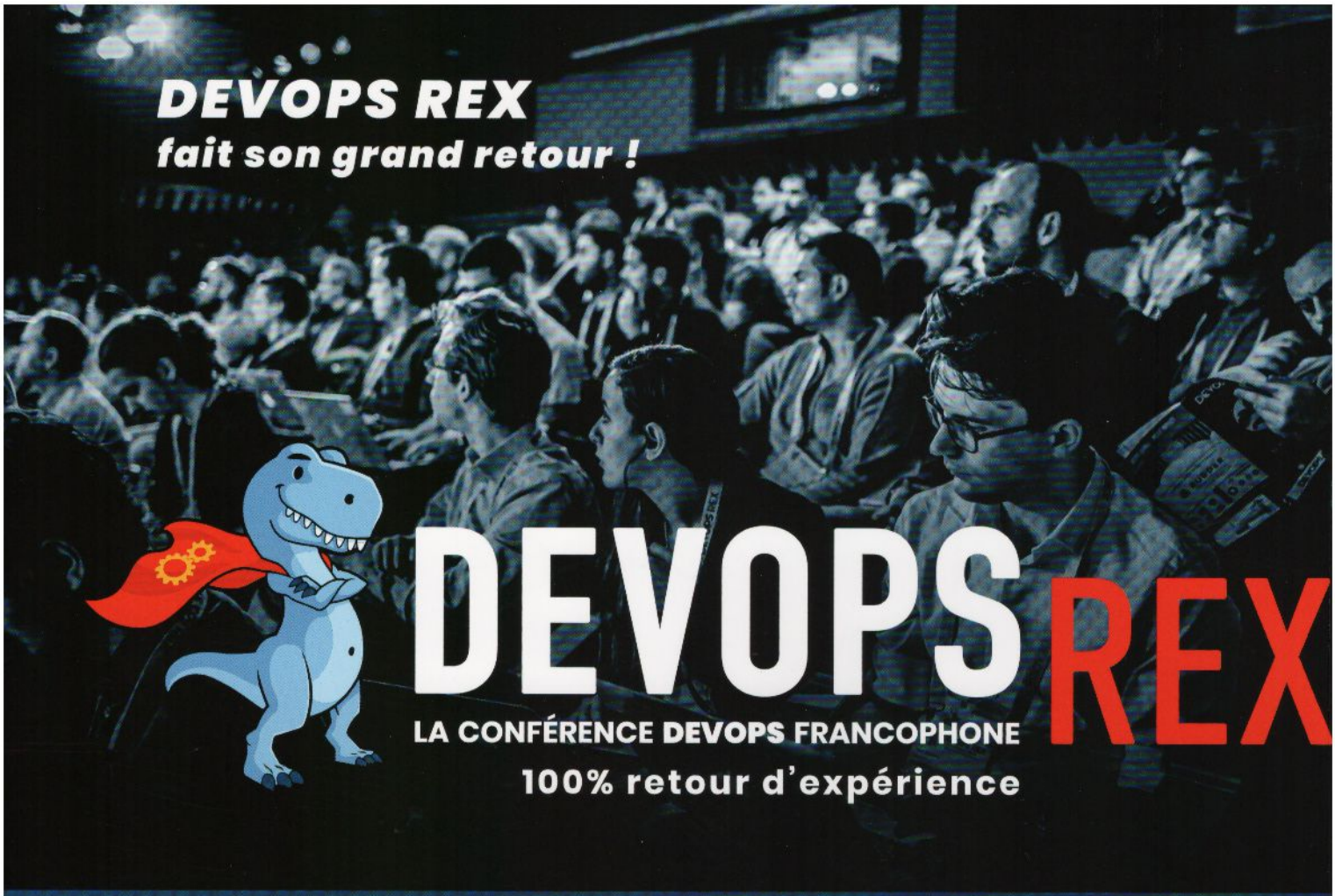
Dernière remarque pour les « anciens » : ne paniquez pas lors de vos tests, ici vous ne payez pas la communication :). Je dis ça, car j'ai souvent eu le réflexe de couper le modem en me disant « ça va coûter cher ». Comme quoi, les vieilles habitudes ont la vie dure... **CP**

RÉFÉRENCES

[CPM] **http://www.columbia.edu/kermit/cpm.html**

[HAYES] **https://www.sparkfun.com/datasheets/Cellular%20Modules/AT_Commands_Reference_Guide_r0.pdf**

[SNIF] **https://embeddedfreak.wordpress.com/2008/08/17/rs232-serial-sniffermonitoring-circuit**



PARIS - 04 & 05 DÉCEMBRE 2024 - PALAIS DES CONGRÈS

Une conférence
sur 2 jours dédiée à l'application du devops en entreprise.

Des témoignages concrets,
100% retours d'expérience, sans placement de produit !

Un espace de solutions,
pour rencontrer des acteurs et networker.

Info et réservation sur
www.devopsrex.fr

Crédit photos : Valentin Napoli – devops Rex 2019

Suivez-nous

Un événement organisé par infoprodigital TRADE SHOWS

Aux mêmes dates et lieu que OPEN SOURCE EXPERIENCE

l'événement Tech – Usages – Business dédié aux solutions IT Open Source.