



Frugalité et conception de circuits pour le traitement du signal audio numérique

Frugality and Circuit Design for Digital Audio Signal Processing

**Maxime Popoff, Romain Michon, Tanguy Risset, Pierre Cochard, Louis Ledoux,
Stéphane Letz, Yann Orlarey, Florent De Dinechin**

INSA Lyon, Inria, Grame-CNCM, CITI, EA3720 69621 Villeurbanne, France
prenom.nom@insa-lyon.fr

Résumé

Les FPGAs (*Field-Programmable Gate Arrays*, ou circuits programmables) sont des plateformes embarquées qui offrent des performances inégalées en termes de latence audio, de capacité de calcul, de gestion d'un grand nombre de canaux audio (entrées/sorties) et de consommation énergétique. La programmation des FPGAs est réputée complexe, notamment à cause de l'utilisation de langages de description de matériel. La synthèse de haut niveau permet toutefois de s'affranchir d'une partie de cette complexité, rendant la programmation des FPGAs plus accessible. Cet article présente l'outil SyFaLa qui permet la compilation de programmes de traitement du signal audio à partir du langage de programmation Faust. Il introduit d'abord la manière dont les plateformes FPGA fonctionnent et les outils de programmation associés. Il illustre quelques réalisations « frugales » utilisant des FPGAs et réalisées grâce à SyFaLa, comme par exemple un système de diffusion WFS (*Wave Field Synthesis*) à bas coût. La compilation de programmes audio sur FPGA permet d'envisager dans un avenir proche la compilation vers des circuits dédiés tels que des ASICs (circuits dédiés) dont le mode de conception est proche de la programmation FPGA. L'article termine par le recensement d'outils et logiciels libres qui permettent d'envisager, à un coût très raisonnable, la mise en place de circuits audio numériques à partir de spécifications haut niveau.

Mots-clés : FPGA, Faust, DSP Audio, Systèmes embarqués, audio spatialisé.

Abstract

FPGAs (*Field-Programmable Gate Arrays*) are embedded platforms offering exceptional performances for audio processing in terms of latency, computational power, and ability to target a large number of audio and inputs and outputs – while consuming less power than conventional digital platforms. FPGA programming is widely recognized as complex, largely because it relies on Hardware Description Languages (HDLs). High-Level Synthesis (HLS) mitigates part of this complexity, thereby lowering the entry barrier and making FPGA design more accessible. This paper presents SyFaLa, a tool enabling the compilation of audio DSP programs written in the Faust programming language. It first provides an introduction to FPGA platforms and their associated programming tools. It then illustrates a few “frugal” implementations achieved with FPGAs using SyFaLa, such as a low-cost WFS diffusion system. Compiling audio programs onto FPGAs also opens the way, in the near future, to



compilation toward dedicated circuits (i.e., ASICs, Application-Specific Integrated Circuits), whose design process is close to FPGA programming. The paper concludes with a survey of open-source design tools that make it possible, at a very reasonable cost, to design audio circuits directly from high-level specifications.

Keywords: FPGA, Faust, Audio DSP, Embedded Systems, Spatial Audio.

1. Introduction

La demande en systèmes audio de haute performance augmente de manière croissante. Les priorités varient grandement en fonction du type d'application visé : puissance de calcul pour la synthèse par modélisation physique (Bilbao, 2009), latence audio pour le contrôle actif d'acoustique (Nelson 1994), nombre de haut-parleurs à gérer pour l'ambisonie (Gerzon, 1973; Zotter and Frank, 2019) ou la Wave Field Synthesis (WFS) (Ziemer, 2018), etc.

Aujourd'hui, la plupart des systèmes audio temps réel reposent sur des processeurs classiques (CPU), des processeurs de signal numérique (DSP), ou sur des microcontrôleurs pour les applications audio embarquées. Cependant, atteindre une latence inférieure à la milliseconde ou gérer des interfaces multicanaux à grande échelle reste un défi majeur avec ces architectures. Cela a conduit à un intérêt croissant pour des plates-formes matérielles spécialisées telles que les FPGAs (*Field-Programmable Gate Arrays*, ou circuits programmables) et les circuits intégrés spécifiques à une application ASICs (*Application-Specific Integrated Circuits*). Les FPGAs représentent une solution prometteuse face à de nombreuses limites rencontrées dans le traitement audio temps réel. Réputés pour leur débit de calcul élevé et leur latence extrêmement faible, ils offrent un moyen de dépasser les contraintes des plates-formes de calcul traditionnelles. Certains systèmes audio industriels ont déjà commencé à tirer parti de la technologie FPGA (ex. tables de mixage numériques, etc.).

Cependant, jusqu'à récemment, le développement de systèmes à base de FPGAs nécessitait une expertise considérable en microélectronique, les rendant largement inaccessibles à la plupart des ingénieurs et développeurs audio. De nouvelles méthodologies de conception matérielle, notamment la synthèse de haut niveau (*High-Level Synthesis*, HLS), ont progressivement réduit le temps et l'expertise nécessaires au développement pour FPGA. Néanmoins, malgré ces améliorations, la programmation des FPGAs reste une tâche complexe. À l'heure actuelle, il n'existe pas de solution largement accessible permettant de programmer des FPGAs directement à partir de spécifications DSP audio de haut niveau de manière simple, sans recourir à des connaissances spécialisées en conception matérielle.

Cet article présente l'outil SyFaLa qui permet la programmation haut niveau des FPGAs pour des applications audio, à partir du langage de programmation Faust (Orlarey, Fober, and Letz, 2004). L'utilisation d'un DSL (*Domain-Specific Language*, ou langage spécifique au domaine de l'audio) réduit significativement l'espace de conception et facilite l'automatisation de la compilation de programmes audio. Cela favorise de nos jours l'émergence de véritables compilateurs pour FPGAs tel que SyFaLa, visant spécifiquement des applications audio.

Pour comprendre la genèse d'un tel outil, nous commençons par présenter les architectures FPGAs et les outils de conception associés (Section 2). Nous présentons ensuite l'outil lui-même (Section 3), sans rentrer trop dans les détails techniques puisque la description de l'outil a déjà été faite par ailleurs (Popoff et al., 2022 ; Popoff, Michon, Risset, Cochard, Letz, Orlarey, and De Dinechin, 2023 ; Michon et al., 2024 ; Cochard et al., 2023 ; Cochard, Popoff, et al., 2024 ; Cochard, Weber, et al., 2024) et il est disponible en libre accès (Emeraude Team, 2025). Nous présentons quelques réalisations en Section 4 qui mettent en avant les avantages que peuvent amener les FPGAs : systèmes audio multi-canaux frugaux en énergie ou en coût, très faible latence etc. Enfin la Section 5 ouvre des perspectives pour aller plus loin que les FPGAs : vers les ASICs. Une communauté de plus en plus nombreuse propose des outils open-source ou des processus à bas coût pour concevoir des circuits dédiés, nous présentons quelques-uns de ces outils et les perspectives vers lesquelles ces derniers pourraient mener.

2. Les architectures FPGAs

Après la Seconde Guerre mondiale, Alan Turing et John von Neumann ont introduit les concepts fondamentaux qui, comme nous le savons aujourd'hui, ont donné naissance aux architectures des ordinateurs utilisés depuis plus de cinquante ans (Lavington, 1980) (voir Figure 1). Le modèle d'architecture proposé par von Neumann, fondé sur une mémoire à accès aléatoire couplée à un processeur, constitue encore aujourd'hui la base de toutes les architectures de CPU. La loi de Moore (Moore, 2006) (selon laquelle le nombre de transistors présents sur une puce de microprocesseur double tous les deux ans, à coût constant) a permis une croissance régulière des capacités de calcul et, avec elle, le développement continu des systèmes numériques.

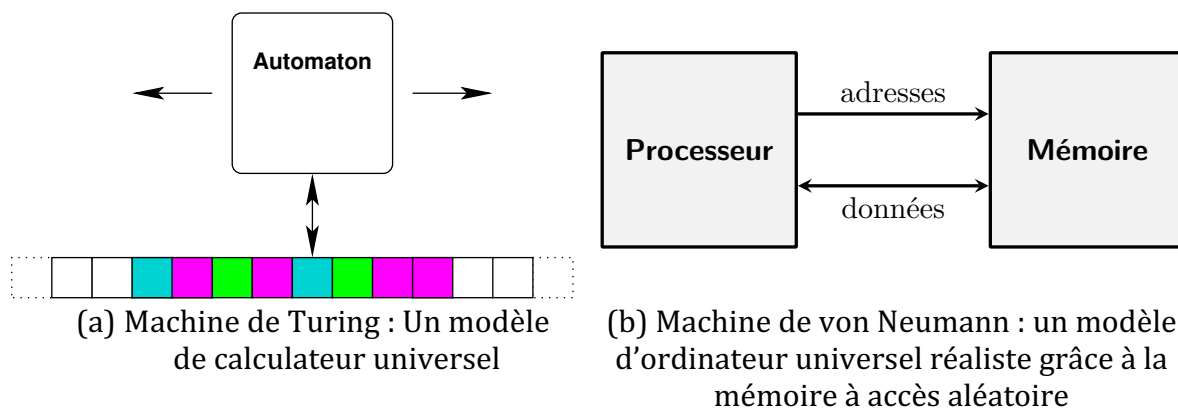


Figure 1 : Les contributions fondamentales de Turing et Von Neumann à l'invention de l'ordinateur.

Aujourd'hui, la loi de Moore n'est plus vérifiée : la taille de gravure des transistors approchant celle de l'atome, les technologies les plus intégrées deviennent extrêmement coûteuses. De nouveaux paradigmes architecturaux apparaissent donc, tels que les FPGA, les GPU (Graphics Processing Units), les NPU (Neural Processing Units), etc. Ces architectures

visent à accroître encore les performances des systèmes numériques, car il ne suffit plus d'attendre simplement la génération suivante de processeurs pour obtenir des gains significatifs.

Pourtant, déjà à l'époque, John von Neumann avait proposé d'autres modèles d'architecture. L'un d'eux, en particulier, l'automate cellulaire (voir Figure 2), a donné lieu à de nombreux développements (Wolfram, 2002). Un automate cellulaire est constitué d'un nombre infini d'automates simples, reliés à leurs voisins — linéairement ou en grille — par des liens de communication.

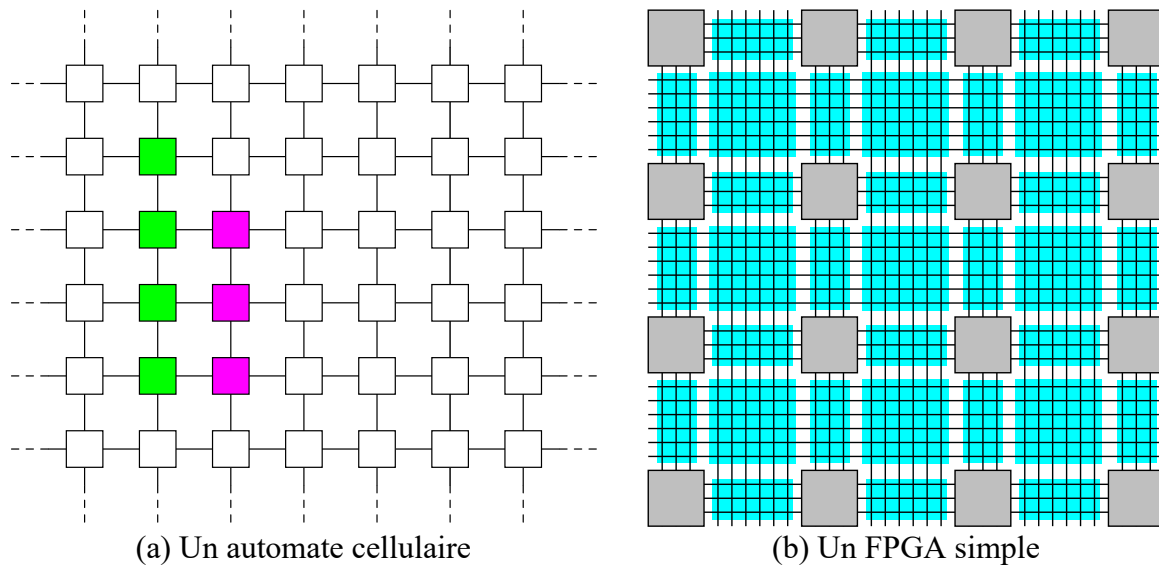


Figure 2 : Un FPGA est constitué de cellules simples reconfigurables reliées entre elles par des liens du largeur de 1 bit. Un FPGA correspond au modèle de l'automate cellulaire, mais avec un réseau de connexions programmables et plus développé (pas seulement des connexions entre voisins).

Les FPGA sont aux automates cellulaires ce qu'un CPU est à une machine de Turing. Un FPGA est constitué d'une grille de cellules reconfigurables reliées entre elles et utilisées pour implémenter aussi bien des fonctions logiques que, plus généralement, n'importe quel circuit séquentiel. Le réseau d'interconnexion entre ces cellules est lui-même reconfigurable et permet de relier un grand nombre de cellules du FPGA. Il faut garder à l'esprit qu'un FPGA contient un grand nombre de ces cellules. Par exemple, le Zynq-7010 de Xilinx, qui est un petit FPGA, en contient environ 28 000.

La structure d'une cellule varie peu d'un constructeur de FPGA à l'autre ; elle est représentée en Figure 3. Une cellule est constituée d'une fonction logique (généralement à 4 entrées d'un bit, parfois à 5 entrées), couplée à un registre d'un bit capable de stocker le résultat de cette fonction. Comme une fonction logique à quatre entrées correspond à 16 combinaisons possibles, elle est implémentée par un simple accès à un tableau de 16 bits chargé dans la cellule lors de la configuration du FPGA. On appelle donc ces cellules des LUT (*Look-Up Tables*), puisqu'elles sont simplement constituées d'une table de bits.

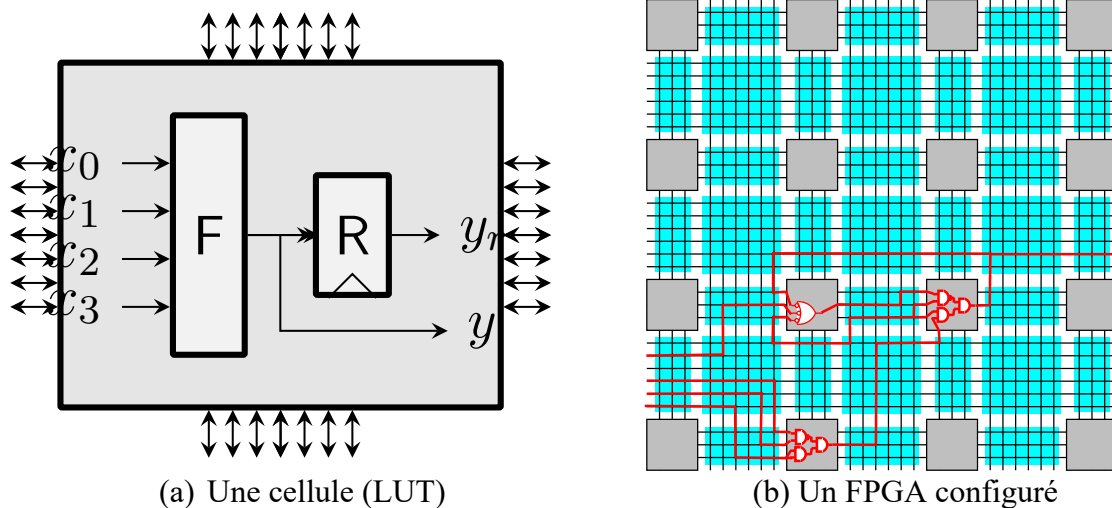


Figure 3 : (a) Une cellule est constituée d'une fonction logique simple associée à un registre de 1 bit. Comme la fonction logique est implémentée par une table, on parle de *Look Up Table* (LUT). (b) Exemple d'un FPGA configuré (*programmé* avec une fonction logique particulière).

Le réseau d'interconnexion est plus complexe, car il est constitué d'un grand nombre d'aiguillages. On dispose ainsi, en quelque sorte, d'un accès aléatoire aux cellules distantes. Il est évidemment impossible de relier directement toutes les cellules entre elles (une cellule n'ayant de toute façon que quatre entrées), mais le concepteur n'a pas à se soucier de ce problème : les outils de placement-routage des FPGA se chargent de transformer un circuit logique en le répartissant sur les cellules et en les connectant grâce au réseau de routage du FPGA.

La configuration d'un FPGA consiste à instancier les LUTs avec les fonctions logiques effectives et à connecter les différentes cellules en positionnant les aiguillages du réseau de routage. La Figure 3, à droite, illustre ce concept : les fils et portes logiques en rouge correspondent à la configuration du FPGA ; certains fils peuvent sortir du circuit, ce sont alors les entrées/sorties qui permettent au FPGA de communiquer avec l'extérieur.

Le cycle de vie d'un FPGA se déroule en deux temps. D'abord, le temps de configuration (entre 1 ms et 1 s), pendant lequel les LUTs sont chargées avec leurs tables de vérité et l'état de chaque aiguillage est défini. Ensuite vient le temps d'exécution (a priori infini) : les données sont traitées par les LUT selon leurs tables de vérité et circulent le long des connexions entre LUT (ces connexions étant statiques une fois le FPGA configuré, voir par exemple Figure 3-b). Elles peuvent être stockées dans les registres des LUT, cadencés par une horloge commune (en général entre 200 et 400 MHz, donc nettement plus lente que celle des CPUs actuels). Durant cette phase, le FPGA se comporte comme un circuit de portes logiques. Un programme pour FPGA correspond donc à un grand nombre de bits de configuration ; on parle de *bitstream* pour le distinguer d'un exécutable pour CPU. Le modèle de programmation d'un FPGA est donc celui du circuit numérique que l'on configure.

Le modèle que nous venons de décrire correspond aux premiers FPGAs apparus dans les années 80. Depuis, de nombreux composants ont été ajoutés sur la puce afin de faciliter leur

programmation. Par exemple, un FPGA contient aujourd’hui de nombreux multiplieurs câblés (habituellement 24 bits en point fixe), appelés DSP, ainsi que de nombreux petits blocs mémoires (de l’ordre de 10 kbits chacun), appelés *block RAM*.

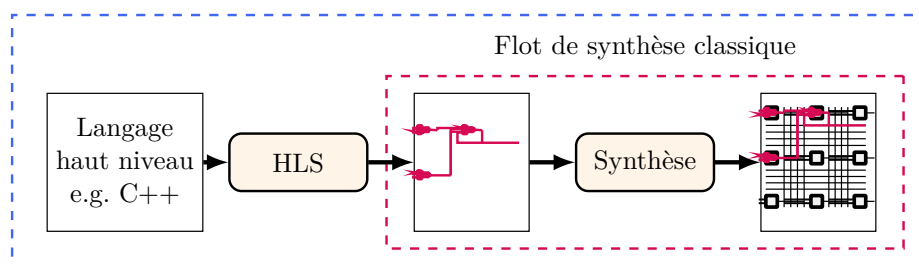
Depuis quelques années, les constructeurs de FPGA ont intégré un composant important sur la puce : un processeur complet, généralement appelé *Processing System*, qui permet d’exécuter n’importe quelle tâche logicielle¹. Il inclut en particulier les bus permettant d’accéder à de la mémoire externe ainsi que les périphériques classiques (UART, USB, I²C, etc.), ce qui facilite grandement l’interface avec le FPGA. Ce *Processing System* communique directement avec la logique reconfigurable via un bus spécifique. L’ensemble FPGA + *Processing System* est généralement désigné sous le terme SoC pour *System on Chip*.

Historiquement, la programmation d’un FPGA ne se fait pas à partir d’un langage de programmation classique, mais à partir d’une description structurée d’un circuit logique exprimée dans des langages de description de matériel tels que VHDL ou Verilog. Une fois cette description réalisée (et validée par simulation), les outils des constructeurs de FPGA effectuent une passe appelée « synthèse », qui transforme la description du circuit en une *netlist*, c’est-à-dire en une liste de portes logiques. Les passes suivantes, appelées « placement » et « routage », répartissent ensuite ces portes logiques sur les LUT. L’ensemble du processus de compilation peut prendre plusieurs heures pour les circuits complexes.

La conception de circuits en VHDL ou en Verilog est une compétence exigeante, plutôt développée dans les formations de microélectronique, mais beaucoup moins en informatique et encore moins en traitement du signal audio. La véritable révolution, depuis environ 2010, est venue de la synthèse de haut niveau (HLS, pour *High-Level Synthesis*), qui permet de partir d’une description écrite dans un langage de programmation, en général un sous-ensemble de C ou C++ (voir Figure 4). La HLS extrait le parallélisme de la spécification séquentielle du programme et conçoit la machine à états qui cadencera le circuit final.

Mais les langages C et C++ ne sont pas réellement des langages haut niveau. Le véritable défi de la compilation de programmes audio vers FPGA est de l’ouvrir à des musiciens ou à des ingénieurs qui ne sont pas nécessairement spécialistes du C++. De nombreux environnements de programmation audio existent et offrent une vision plus abstraite que celle des langages de programmation généralistes. Nous avons choisi le langage Faust parce que nous maîtrisons son compilateur, mais ce travail pourrait sans doute être réalisé à partir d’autres langages comme Pure Data ou Csound.

Flot de synthèse avec HLS



¹ Le terme « logiciel » est ici à mettre en opposition avec le concept de « matériel ». Dans ce cas, le logiciel est exécuté sur le *Processing System* et le matériel est implémenté sur le FPGA.

Figure 4 : Le flow de compilation HLS utilisant le flot de conception classique pour générer les *bitstreams*

La Figure 5 illustre l’objectif visé : un environnement de synthèse vraiment haut niveau, permettant de partir d’un DSL audio, de générer du C++, puis d’utiliser ensuite le flot HDL classique. La section suivante décrit cette réalisation à travers le projet SyFaLa, qui utilise le langage Faust comme DSL audio.

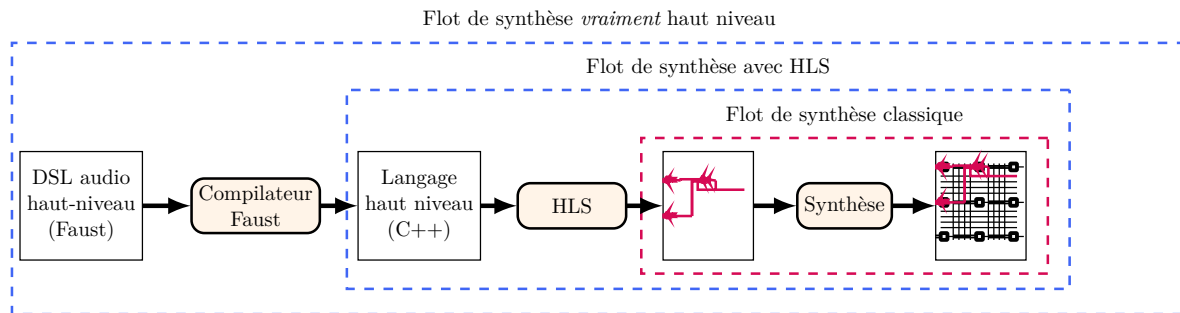


Figure 5 : Le flot de synthèse utilisé dans SyFaLa en partant de programmes Faust

3. SyFaLa: un compilateur audio pour FPGA

Cette section présente un flot de compilation entièrement automatisé, allant d’un langage de haut niveau pour le traitement du signal audio (Faust) vers une co-implémentation FPGA/CPU. Ce flot de compilation, nommé SyFaLa, permet la production directe de son sur des cartes FPGA équipées d’un codec audio. Il a déjà été partiellement présenté dans les publications précédentes (Popoff et al., 2022 ; Popoff, Michon, Risset, Cochard, Letz, Orlarey, and Dinechin, 2023 ; Cochard et al., 2023). Le flot utilise l’outil de synthèse standard de Xilinx (*vitis_hls* et *vivado*) comme *back-end* et le compilateur Faust comme *front-end*. Hormis les outils propriétaires Xilinx, il est entièrement *open-source* et disponible sur GitHub (Emeraude Team, 2025).

Faust est un langage de programmation fonctionnel dédié à la synthèse sonore et au traitement audio, avec un accent particulier sur la conception de synthétiseurs, d’instruments de musique et d’effets sonores. L’un de ses principaux atouts réside dans son compilateur, capable de produire un code optimisé pour une large gamme de langages cibles (C++, C, LLVM IR, Rust, WebAssembly (Letz, Orlarey, and Fober, 2018), plug-ins Web Audio (Ren et al., 2020), etc.) à partir d’un unique programme Faust. Par ailleurs, Faust se distingue par son adéquation au traitement audio embarqué, grâce à son support natif de nombreuses plateformes matérielles pouvant produire de l’audio (Michon et al., 2020), ce qui en fait un candidat naturel pour le déploiement sur architectures FPGA.

SyFaLa est un flot de compilation qui combine le compilateur Faust avec un outil de synthèse de haut niveau (*vitis_hls*), lequel joue le rôle de *middle-end* dans la compilation en optimisant l’implémentation obtenue pour les cibles FPGA. Cette combinaison est rendue possible par le langage C++, qui est à la fois l’un des points de sortie du compilateur Faust, et le point d’entrée de *vitis_hls*. Le code Faust haut niveau écrit par l’utilisateur passe ainsi par



différentes phases de traductions et d'optimisations, pour arriver à une description matérielle qu'il sera possible de porter sur une cible FPGA.

L'une des optimisations clés réalisées par le compilateur Faust consiste en la séparation des calculs en fonction de leur nature et de leur criticité. On distingue effectivement dans le domaine de l'audio numérique des calculs dits « critiques », indispensables à la génération et au traitement du signal audio, qui sont effectués à chaque échantillon, et des calculs dits « de contrôle », dont la source provient des actions de contrôle effectuées par l'utilisateur, et pour lesquelles la fréquence d'échantillonnage peut être bien plus réduite. Il faut enfin ajouter à cela les calculs qui ne seront effectués qu'une seule fois, au démarrage du programme, afin d'initialiser certaines données en mémoire.

Ces optimisations sont d'une grande importance dans le contexte d'une implémentation sur FPGA. Ceux-ci étant limités en ressources, il est important de les utiliser le plus efficacement possible. Cela signifie privilégier leur utilisation pour des calculs continus et à haute fréquence, afin de maximiser leur rendement. À titre d'illustration, une fonction sinus/cosinus (très coûteuse en termes de ressources FPGA) qui ne serait utilisée qu'une seule fois à l'initialisation du programme afin de calculer certains coefficients constants serait une utilisation particulièrement inefficace de ces ressources.

Par conséquent, grâce à la compartementalisation effectuée par le compilateur Faust, il est possible de déplacer les calculs d'initialisation ou de contrôle pour qu'ils soient traités par le *processing system* du SoC (le CPU), par le biais d'un programme externe tournant en parallèle du circuit FPGA. Les résultats de ces calculs resteront toutefois accessibles à ce dernier grâce à un bus mémoire de type AMBA, utilisant différents protocoles, comme AXI ou AXI-Lite, assurant la bonne communication entre les deux entités CPU et FPGA.

Concrètement, pour réaliser cette séparation FPGA/CPU, le compilateur Faust va générer deux implémentations C++ différentes à partir d'un même programme d'entrée, comme l'illustre la Figure 6. L'une d'entre elles sera à destination du FPGA, et s'occupera de gérer les calculs DSP « critiques », tandis que l'autre sera à destination du CPU du SoC, et sera chargé d'assurer les calculs de contrôle et d'initialisation.

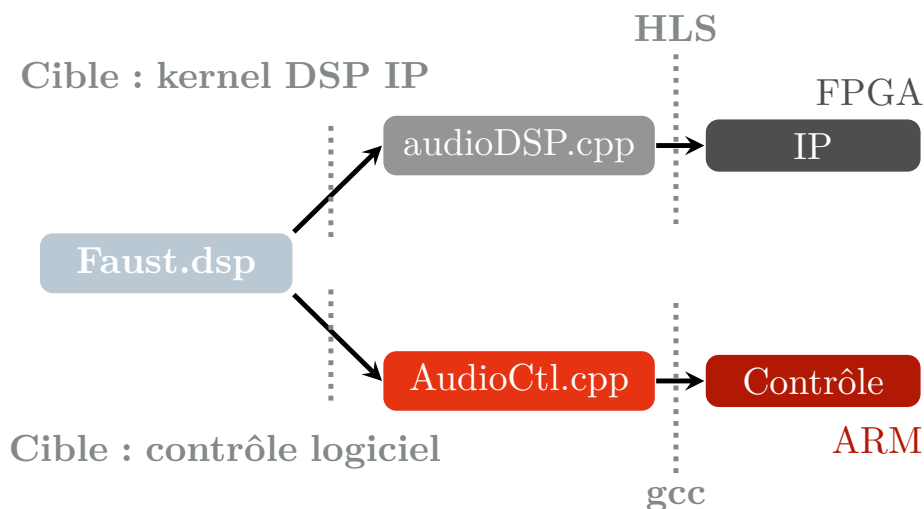
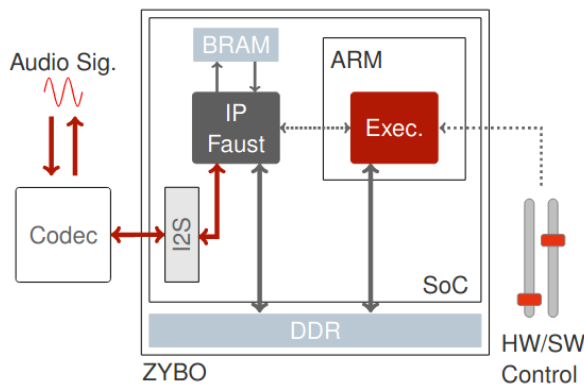


Figure 6 : Vue d'ensemble du compilateur SyFaLa de Faust vers FPGA faisant appel au compilateur Faust deux fois.

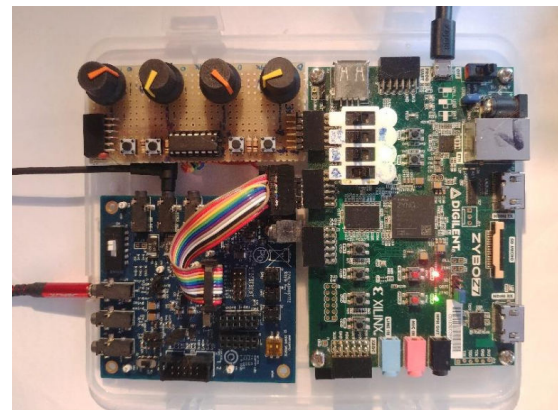
Une fois un important travail d’interfaçage réalisé (Popoff et al., 2022; Popoff, Michon, Risset, Cochard, Letz, Orlarey, and Dinechin, 2023 ; Cochard et al., 2023), le système issu du flot de compilation présenté à la Figure 6 et déployé sur FPGA est illustré à la Figure 7. La Figure 7-a montre l’architecture du système, déployée sur une carte Zybo mais facilement adaptable à toute carte FPGA connectée à un codec, tandis que la Figure 7-b présente un premier exemple d’instrument réalisé avec un FPGA (carte Zybo) et le codec Analog Devices ADAU178

Le programme du Listing 1 est un exemple de programme Faust : un oscillateur sinusoïdal implémenté avec un filtre biquad (Smith 2007), et adapté à une mise en œuvre sur FPGA. « Adaptée » signifie dans ce contexte que le code n’appelle pas la fonction `sin` à chaque calcul d’échantillon : les appels aux fonctions `sin` et `cos` sont effectués au *control rate*, c’est-à-dire uniquement lorsque l’utilisateur modifie la fréquence de l’oscillateur. Implémenter la fonction `sin` de la bibliothèque mathématique C directement sur FPGA serait extrêmement coûteux en termes de ressources. De la même manière, cette spécification n’utilise pas de table pour stocker les valeurs de la forme d’onde, ce qui nécessiterait une mémoire externe. Une telle approche serait possible, mais limiterait le nombre de sinusoïdes implémentées simultanément sur le FPGA.

La fonction présentée dans le Listing 1 a été implémentée dans la bibliothèque standard de Faust par Julius Smith (Grame-CNCM, 2025 ; Smith, 2007). Le paramètre `freq`, réglé par l’utilisateur, permet de configurer un filtre résonnant simple ou biquad (la fonction `n1f2`) qui génère la sinusoïde à la fréquence voulue.



(a) Système audio résultant de la compilation SyFaLa



(b) Premier prototype SyFaLa : Le Popophone (Popoff et al., 2022)

Figure 7 : (a) Le système audio complet est interfacé à un codec audio (directement depuis le FPGA) et à un outil de contrôle matériel ou logiciel (en passant par le *Processing system*). (b) le Popophone utilise un codec optimisé pour la faible latence et est contrôlé par 4 potentiomètres.



```
import("stdfaust.lib");

freq = hslider("freq [knob:1]",440,50,1000,0.01);
nlf2(f,r,x) = ((_<:_,_),(_<:_,_) :
              (*(s),*(c),*(c),*(0-s)) :>
              *(r),+(x))) ~ cross
with {
  th = 2*ma.PI*f/ma.SR;
  c = cos(th);
  s = sin(th);
  cross = _,_ <: !,_,_ !;
};

impulse = 1-1';
process = impulse : nlf2(freq,1) : !,_ <: _,_;
```

Listing 1 : Exemple de programme Faust sine.dsp.

Avec la chaîne d'outils SyFaLa, le programme sine.dsp peut être directement compilé vers un FPGA à l'aide d'une simple invocation en ligne de commande (voir Listing 2). Le programme est entièrement déployé sur une carte Zybo Z20 en environ 15 minutes de compilation sur un PC (temps de construction initial, principalement dû à la synthèse vivado) et peut être contrôlé, par exemple, via la connexion USB entre la carte Zybo et le PC. Le processus est entièrement automatisé et ne nécessite aucune connaissance en conception matérielle de la part de l'utilisateur.

```
syfala.tcl ./sine.dsp -b Z20
```

Listing 2 : Ligne de commande SyFaLa pour compiler intégralement l'exemple sine.dsp sur le FPGA Zybo Z20

Les deux listings suivants montrent des extraits des codes générés (c'est-à-dire les fichiers audioCtl.cpp et audioDSP.cpp de la Figure 6). Le Listing 3 illustre une partie du code du *Processing System* : on observe que les fonctions sin et cos sont appelées sur le processeur ARM, et que les résultats de ces fonctions sont ensuite transmis au FPGA via le tableau fControl, qui est envoyé sur le bus AXI reliant le processeur et le FPGA. On remarque également que le contrôleur freq est utilisé (dsp->fHslider0). La manière dont ce contrôle est interfacé avec l'utilisateur n'est pas détaillée ici : on peut utiliser un potentiomètre, comme sur le Popophone de la Figure 7, ou une interface musicale (ex. MIDI, OSC) grâce à Linux, comme expliqué ci-dessous.

```
[... in audioCtl.cpp ...]
void controlmydsp(mydsp* dsp, int* iControl, float* fControl,
                 int* iZone, float* fZone) {
    fControl[0] = (dsp->fConst0 * (float)dsp->fHslider0) ;
```



```
fControl[1] = sinf(fControl[0]) ;  
fControl[2] = cosf(fControl[0]) ;  
}
```

Listing 3 : Sine example, excerpt of program audioCtl.cpp

Le listing 4 est un extrait du code destiné à être donné à la HLS pour produire l'IP² audio du FPGA (le IOTA0 étant une optimisation utilisée par le compilateur de Faust pour implémenter efficacement les buffers circulaires). On observe que le tableau fControl est récupéré en entrée de la fonction, et que le code de cette fonction correspond au filtre biquad (fonction n1f2 du Listing 1). Ce code réalise le calcul d'un seul échantillon, car SyFaLa est par défaut configuré en mode *one sample* : le FPGA ne bufferise pas les échantillons, il les traite un par un, ce qui permet d'obtenir la latence la plus faible possible. Il existe également un mode *multi-sample*, qui permet d'optimiser les accès à la mémoire externe lorsqu'elle est utilisée.

```
[... in audioDSP.cpp ...]  
void computemydsp(mydsp* dsp, FAUSTFLOAT* inputs,  
    FAUSTFLOAT* outputs, int* iControl, float* fControl,  
    int* iZone, float* fZone) {  
    dsp->iVec0[(dsp->IOTA0 & 1)] = 1 ;  
    float fTemp0 = dsp->fRec1[((dsp->IOTA0 - 1) & 1)] ;  
    float fTemp1 = dsp->fRec0[((dsp->IOTA0 - 1) & 1)] ;  
    dsp->fRec0[(dsp->IOTA0 & 1)] = ((fControl[1]*fTemp0) +  
        (fControl[2] * fTemp1)) ;  
    dsp->fRec1[(dsp->IOTA0 & 1)] = (((float)(1 -  
        dsp->iVec0[((dsp->IOTA0 - 1) & 1)]) + (fControl[2] *  
        fTemp0)) - (fControl[1] * fTemp1)) ;  
    float fTemp2 = dsp->fRec1[((dsp->IOTA0 - 0) & 1)] ;  
    outputs[0] = (FAUSTFLOAT)fTemp2 ;  
    outputs[1] = (FAUSTFLOAT)fTemp2 ;  
    dsp->IOTA0 = (dsp->IOTA0 + 1) ;  
}  
[...]
```

Listing 4 : Sine example, excerpt of program audioDSP.cpp

Depuis les premiers résultats de SyFaLa, l'outil a été amélioré. SyFaLa peut désormais prendre une spécification en C++ en entrée (pour contrôler plus facilement le code source), et la distribution Linux installée sur le *Processing System* permet d'interfacer aisément le FPGA via Ethernet.

La Figure 8 illustre de manière exhaustive tous les fichiers générés par la commande SyFaLa en 2025. La cible technologique est limitée aux architectures Xilinx comportant un

² Dans notre contexte, le terme *IP* (*Intellectual Property*) est utilisé pour désigner un circuit électronique

SoC. L'ensemble du processus étant automatique, il ne requiert aucune expertise de la part de l'utilisateur. Évidemment, si le programme est trop volumineux ou trop complexe, il ne pourra pas tenir sur le FPGA ou ne sera pas traité suffisamment rapidement. Dans ce cas, une réécriture du programme source sera nécessaire afin de mieux contrôler l'utilisation des ressources lors de la compilation par `vitis_hls`. Des compétences avancées en programmation FPGA seront alors requises.

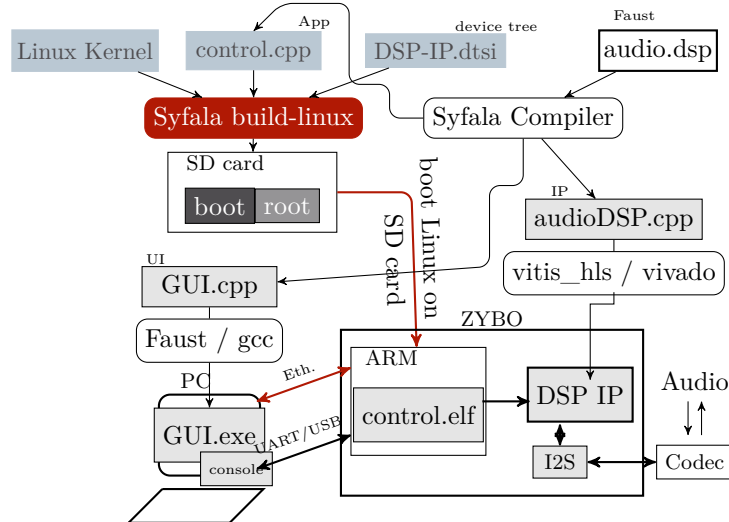


Figure 8 : L'ensemble des fichiers générés (en gris clair) par SyFaLa à partir d'un fichier Faust `audio.dsp`.

4. Résultats expérimentaux

Cette section présente quelques résultats obtenus grâce à la chaîne de compilation SyFaLa. Deux domaines principaux se prêtent à l'utilisation d'une chaîne de compilation audio pour FPGA : les applications temps-réel nécessitant une très faible latence, telle que le contrôle acoustique, et les applications demandant un grand nombre de canaux, comme les technologies audio immersives. Dans les deux cas, les FPGA offrent une solution plus économique et moins énergivore que du matériel spécialisé présentant des performances équivalentes.

La section 4.1 présente les résultats obtenus concernant les performances de latence *analog-to-analog* atteintes par les programmes compilés avec SyFaLa. La section 4.2 illustre quant à elle un exemple de système de son immersif frugal (c'est-à-dire peu coûteux), réalisé grâce à la technologie SyFaLa.

4.1. Latence ultra-faible avec SyFaLa

La latence d'un système audio doit être mesurée entre l'entrée analogique et la sortie analogique du système. En effet, plusieurs sources de latence sont à prendre en compte. Il y a bien sûr la latence de l'IP audio compilée, qui, dans notre cas, est limitée à la durée d'un échantillon en mode *one sample*. Mais il faut aussi considérer la latence introduite par le codec audio ainsi que par l'interface I2S. Concernant cette dernière, il est possible de réaliser une implémentation de l'I2S sur le FPGA dont la latence est également limitée à un échantillon. Une fréquence d'échantillonnage plus élevée conduit donc à une latence plus faible.



Concernant la latence introduite par le codec audio, elle dépend de nombreux paramètres liés à la fois à la qualité du codec et à sa configuration. Nous avons mesuré cette latence avec trois codecs audio, dont deux réputés pour leur faible latence. Ces codecs disposent d'une horloge interne utilisée pour les traitements appliqués au signal (généralement des filtres simples). On distingue donc deux fréquences de référence : la fréquence de calcul interne du codec (Ck^{ADC}) et la fréquence d'échantillonnage (f_s^{I2S}).

Les trois codecs testés sont :

- Le codec *Analog Devices ADAU1777* ($\max f_s^{I2S} = 192 \text{ kHz}$ & $\max Ck^{ADC} = 768 \text{ kHz}$).
- Le codec *Analog Devices ADAU1787*, qui est celui qui fournit les meilleures performances en termes de latence ($\max f_s^{I2S} = \max Ck^{ADC} = 768 \text{ kHz}$).
- Le codec *Analog Devices SSM2603*, qui est intégré sur la carte Zybo, et qui n'est pas optimisé pour la latence ($\max f_s^{I2S} = \max Ck^{ADC} = 96 \text{ kHz}$).

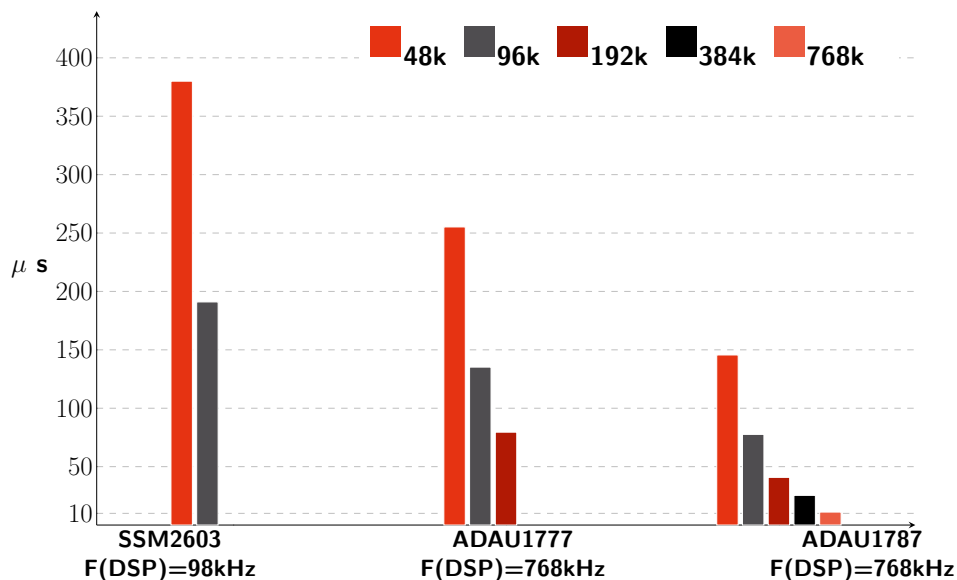


Figure 9 : Latence *analog to analog* obtenue pour un programme audio compilé avec SyFaLa pour différents codecs audio et à différentes fréquences d'échantillonnages.

Les résultats de latence, extraits de (Popoff et al., 2022), sont présentés à la Figure 9. La meilleure latence ($11.1\mu\text{s}$) est obtenue avec le codec le plus rapide, à une fréquence d'échantillonnage de 768kHz. Avant ces travaux, la plus faible latence mesurée était de $180\mu\text{s}$, rapportée dans (Vannoy et al., 2019) avec un codec cadencé à 768kHz. Grâce aux FPGAs, on gagne donc un ordre de grandeur et l'on peut envisager des applications capables de réagir beaucoup plus rapidement aux stimuli entrants.

4.2. Dispositif de son immersif avec SyFaLa

Les systèmes audio immersifs modernes exploitent plusieurs dizaines, voire plusieurs centaines, de canaux audio. La chaîne de compilation SyFaLa s'adapte automatiquement au nombre de canaux de l'application audio compilée. Le contrôle I2S peut être configuré en mode

TDM (*Time Division Multiplexing*) pour prendre en charge un grand nombre de canaux (par exemple, 256 sur une carte Zybo-Z20). Cela a permis, par exemple, la mise au point d'un système WFS (Ziemer, 2018) utilisant une carte audio équipée de 32 codecs, comme décrit dans (Popoff, Michon, and Risset, 2024).

Le système résultant, la *Space Bar* (voir Figure 10), a été présenté à la conférence NIME en 2023 (Michon et al., 2024). Il propose un dispositif WFS à 32 canaux, programmable en Faust, capable de diffuser plusieurs sources avec des localisations virtuelles différentes, contrôlables à l'aide d'un joystick.

Le coût total de fabrication de la *Space Bar* est inférieur à 1000 €. Bien entendu, elle a été réalisée avec des codecs de qualité inférieure à ceux mentionnés dans la section précédente, mais la barre a démontré une qualité sonore suffisante pour un rendu efficace de l'audio spatialisé (Quiédeville et al., 2025). Une œuvre musicale dédiée à cette barre a été réalisée en collaboration avec Grame-CNCM par le compositeur Frédéric Kahn³.



Figure 10 : La *Space Bar*, dispositif WFS frugal exposé au *showroom* d'Inria Grenoble

5. Pour aller plus loin: conception frugale de circuits audio

Les FPGAs offrent une plateforme flexible pour prototyper rapidement des architectures numériques (voir §2), mais cette flexibilité repose sur une logique reconfigurable coûteuse en énergie et en surface.

5.1. ASICs: Anima Sana In Corpore Sano

À l'inverse, les ASICs, de leur vrai acronyme *Application-Specific Integrated Circuits* (où l'on pourrait voir dans la « spécificité » une forme de santé), implémentent de manière fixe les fonctions matérielles, permettant des gains significatifs en efficacité énergétique, en performance et en densité d'intégration. Un circuit ASIC peut atteindre un débit supérieur avec une consommation moindre, mais il perd la capacité de reconfiguration qui fait la force des FPGAs. Dans le domaine de l'audio, on trouve depuis longtemps des ASICs spécialisés, qu'il s'agisse des convertisseurs numériques-analogiques (DAC) et analogiques-numériques (ADC), ou de processeurs de codecs intégrés. Certains instruments reposent également sur des ASICs

³ <https://www.fetedelascience.fr/la-barre-d-espace-ou-comment-creer-son-espace-acoustique-en-temps-reel>



de calcul audio : le synthétiseur Yamaha DX7 (Shirriff, 2021) utilisait déjà dans les années 1980 un circuit spécifique pour la synthèse FM, et de nombreux synthétiseurs modernes intègrent encore des DSP propriétaires (Analog Devices SHARC, Motorola 56300, etc.) optimisés pour les traitements en temps-réel. Ces puces dédiées démontrent l'intérêt de l'approche : consommer moins, occuper moins d'espace et atteindre une qualité audio supérieure grâce à un circuit pensé dès l'origine pour cette fonction.

Cependant, la conception et la fabrication d'un ASIC ont longtemps été hors de portée pour les chercheurs académiques ou les petites structures, et encore moins envisageable pour un hautboïste travaillant dans sa chambre. L'étape ultime de ce processus, le *tapeout*⁴, correspond à la transmission de la description physique complète du circuit, sous la forme de fichiers GDS (*Graphic Database System*), afin de réaliser les masques de silicium prêts à être gravés. Jusqu'à récemment, l'ensemble du processus de conception et de fabrication d'une puce, depuis la spécification jusqu'au *tapeout*, représentait un investissement de plusieurs millions d'euros. Sans compter la mobilisation d'outils propriétaires complexes et des cycles de développement pouvant s'étendre sur plusieurs années. La situation évolue toutefois rapidement grâce à un ensemble d'initiatives open-source qui visent à démocratiser la microélectronique, en réduisant drastiquement les coûts et en proposant des outils ouverts.

5.2. Silicium et frugalité

Heureusement, les temps changent, et avec eux la manière dont la microélectronique devient accessible. L'un des moteurs de ce renouveau est le principe des *Multi-Project Wafers* (MPW). Un *wafer* est une plaque de silicium sur laquelle sont gravés simultanément plusieurs circuits intégrés, puis découpés en unités individuelles appelées *dies*. Plutôt que de réserver un *wafer* entier à un seul design, plusieurs projets partagent la même surface, ce qui réduit les coûts de plusieurs ordres de grandeur. Les grands industriels font l'inverse, en produisant des *wafers* entiers avec un seul et même design⁵. Le modèle des MPW, fondé sur le partage, rend aujourd'hui possible la fabrication d'une puce fonctionnelle pour quelques centaines de dollars. Ces campagnes de fabrication mutualisées sont appelées *shuttles*, et elles constituent la pierre angulaire de l'essor des ASICs open-source. La Figure 11 illustre ce principe à travers différents niveaux de visualisation. À gauche, on observe un slot de MPW qui, en plus de regrouper plusieurs projets sur un même wafer, contient lui-même plusieurs sous-designs rassemblés dans une unique soumission. L'exemple central correspond à l'un de nos propres travaux (Ledoux and Casas, 2022, 2023). Il a également été présenté dans une interview avec Matt Venn⁶, figure emblématique du mouvement open silicon depuis ses débuts, dont le travail de vulgarisation et de formation (TinyTapeout, cours en ligne, tutoriels) a joué un rôle clé dans la diffusion de ces pratiques.

4 Le terme *tapeout* vient de l'époque où les fichiers GDS étaient enregistrés sur des bandes magnétiques (*tapes*) puis expédiés physiquement à la fonderie (*out*), parfois littéralement transportés sur un chariot.

5 Après découpe et tests, les *dies* sont triés selon leurs performances réelles, influencées par leur position sur le *wafer*. Ce tri, appelé *binning*, est ensuite exploité pour segmenter la gamme commerciale, comme dans le cas des processeurs Intel Core i3, i5 et i7.

6 https://youtu.be/0tv-nae4YxM?si=xy7_BZyhmO7kztk



Le véritable tournant est venu en 2020, lorsque Google s'est allié à la fonderie SkyWater pour lancer des *shuttles* financés permettant la fabrication gratuite de designs *open-source* (Google and SkyWater 2020). Pour la première fois, un PDK (*Process Design Kit*) industriel a été rendu accessible à tous : le Sky130, distribué sur GitHub et utilisable directement via un simple clonage⁷. Un PDK décrit les règles de fabrication, les modèles de transistors, les bibliothèques standard et les vues de simulation. Avant cela, de nombreux PDKs restaient strictement confidentiels et protégés par des NDA (*Non-Disclosure Agreements*). L'ouverture de Sky130 a permis à des laboratoires, étudiants et amateurs d'accéder aux fondations de la microélectronique.

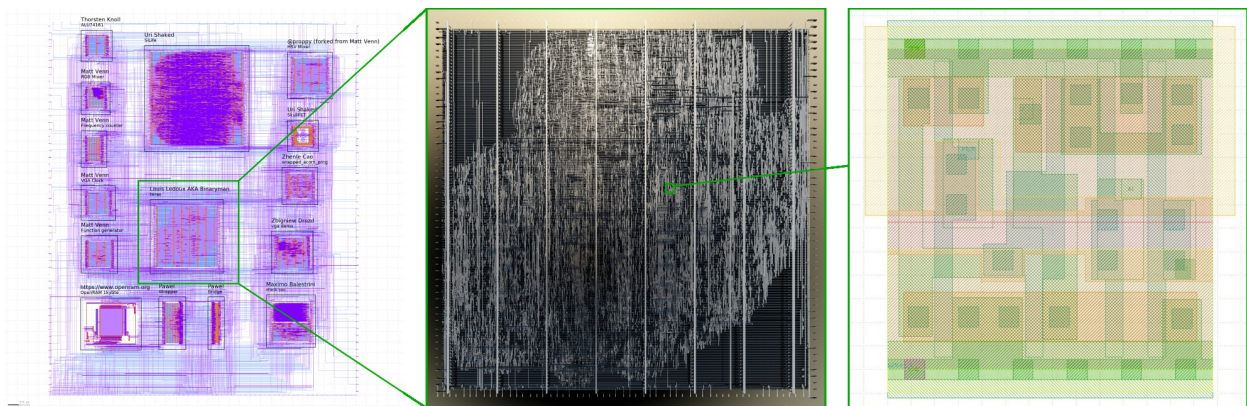


Figure 11 : Exemple de MPW. À gauche : vue *GDSII* d'un slot de MPW contenant plusieurs sous-designs soumis ensemble. Au centre : zoom sur l'un de ces projets. À droite : zoom sur une cellule standard (*a21boi*), où l'on distingue l'agencement des couches technologiques (diffusion, polysilicium et interconnexions métalliques) formant les transistors.

Cette initiative a suscité une vague d'enthousiasme : des dizaines de projets académiques, artistiques ou amateurs ont été lancés via les *shuttles* soutenus par Google et la plate-forme Efabless, qui joue un rôle clé dans la mise à disposition d'outils et dans la coordination logistique des campagnes de fabrication. En parallèle, des outils libres se sont imposés : OpenROAD (flux complet PDK-agnostic (T. Ajayi et al., 2019 ; Tutu Ajayi et al., 2019)), et OpenLane (Shalan and Edwards, 2020), construit sur OpenROAD et adapté à Sky130, qui permet de transformer un circuit décrit en Verilog en un fichier GDSII prêt à être fabriqué.

À une autre échelle, des projets comme *TinyTapeout* poussent la logique de mutualisation encore plus loin. Ils regroupent plusieurs centaines de micro-designs sur un seul *shuttle*, permettant à chacun de recevoir un ASIC accompagné d'un PCB (*Printed Circuit Board*) fonctionnel et prêt à l'emploi pour moins de 200 \$. Cette démarche pédagogique et communautaire a déjà attiré de nombreux étudiants, enseignants et amateurs curieux de franchir le pas du silicium. Ainsi, ce qui relevait hier d'une industrie fermée est devenu aujourd'hui un terrain d'expérimentation ouvert et abordable.

⁷ <https://github.com/google/skywater-pdk>



Ces initiatives transforment profondément la recherche et la formation en microélectronique. L'ouverture des PDKs, la mutualisation des coûts et la disponibilité d'outils libres créent un écosystème inédit reliant informatique et électronique. L'Europe accompagne ce mouvement par des actions structurantes, comme la lettre ouverte signée par de nombreuses universités appelant à développer une infrastructure académique autour du silicium libre⁸. Par ailleurs, des institutions comme l'ETH Zürich ont déjà lancé des cours de microélectronique open-source, avec le projet VLSI de leur département qui propose un cursus via des outils open source et un PDK ouvert⁹. De nouveaux cursus commencent ainsi à inclure la conception ASIC dans les formations en informatique et traitement du signal. Dès lors, concevoir un ASIC audio spécifique devient une option réaliste, non seulement pour les laboratoires mais aussi pour les artistes et les hautboïstes.

5.3. Quand le silicium devient musique

Les initiatives autour des ASICs *open-source* ne se limitent pas aux usages industriels : la création sonore a trouvé dès les premières éditions de *TinyTapeout* un terrain d'expérimentation privilégié. Plusieurs projets ont exploré la synthèse et la génération audio matérielle à très faible coût. Parmi eux, *PWM Audio* (Meng, 2023) convertit un flux numérique 8 bits en signal analogique via modulation de largeur d'impulsion, tandis que *ChipTune* (Everest, 2023) implémente un générateur d'ondes carrées rappelant les puces sonores des consoles 8 bits. Certains concepteurs ont choisi de réimplémenter des circuits historiques tels que le générateur AY-3-8913 (Zioma, 2024b), ou encore le SN76489 (Zioma, 2023), utilisés dans de nombreux ordinateurs et consoles des années 1980. Du côté matériel, des extensions comme *tt-audio-pmod* (Bell, 2023) permettent de brancher directement une sortie PWM sur un casque ou un haut-parleur, transformant ces prototypes en instruments sonores exploitables. On peut également citer *36 Sine Synth* (Edwards, 2023), qui génère des sinusoïdes simples pilotées par les entrées PMOD, ainsi que le *Sine Wave Synthesizer (SWS)* (Scherzer, 2024), qui implémente une approximation matérielle d'onde sinusoïdale. Dans une approche plus ambitieuse, *AudioChip_V2* (Knoll, 2024), conçu par Thorsten Knoll, génère différentes formes d'onde via des sorties PWM et prévoit des entrées de contrôle compatibles avec l'univers des instruments modulaires. Certains membres du projet envisagent même de transformer ce circuit en un module Eurorack complet, prolongeant ainsi l'expérimentation *open-source* jusque dans la lutherie électronique.

Un autre projet marquant est *Drop* (Zioma, 2024a), issu de la scène *demoscene* et présenté à ORConf 2024. Il s'agit d'une démonstration artistique implantée sur une puce fabriquée via un *shuttle* libre (*open-source*), combinant graphismes animés et sortie audio dans un espace matériel extrêmement restreint. Par sa frugalité et son inventivité, ce type d'expérience montre que même dans des contextes où les moyens sont extrêmement limités, les infrastructures ouvertes rendent possible la réalisation de puces intégrant à la fois du graphisme et de l'audio.

8 <https://open-source-chips.eu/>

9 https://vlsi.ethz.ch/wiki/Main_Page

C'est ainsi que ces réalisations montrent que la mutualisation via *TinyTapeout* rend possible des expériences sonores qui, autrefois, auraient nécessité des moyens industriels considérables. De simples générateurs PWM aux réimplémentations de circuits historiques, jusqu'aux démonstrations artistiques comme *Drop*, les projets *TinyTapeout* témoignent de la diversité des approches sonores possibles sur ASICs open-source. Ainsi, l'exploration sonore devient une composante à part entière du mouvement plus large du silicium ouvert, où la créativité ne se limite plus au code mais s'inscrit directement dans la matière de la puce.

L'ouverture artistique ne se limite pas aux ondes sonores, elle s'étend également aux formes et aux ondes visuelles révélées par les circuits eux-mêmes. Les fichiers physiques produits lors du placement-routage (*GDSII*) révèlent des structures géométriques d'une grande richesse visuelle. Des outils récents comme *ArtistIC* (Benz et al., 2025) exploitent ces motifs pour générer des rendus photoréalistes et artistiques de puces, illustrant le potentiel visuel du silicium. Indépendamment de ces approches, nous avons réalisé nos propres rendus *ray tracing* de circuits, en façonnant la topologie de la puce selon un logo conçu par notre équipe, mêlant ainsi rigueur scientifique et esthétique graphique (voir Figure 12).

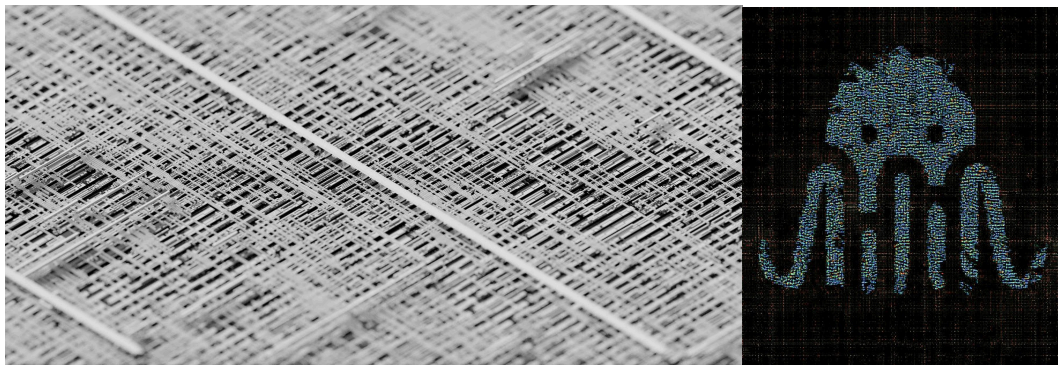


Figure 12 : La microélectronique ouverte comme support artistique. Gauche : rendu *ray tracing* d'une puce conçue avec OpenROAD, avec un jeu sur les matériaux où le métal est remplacé par du verre. Droite : puce fonctionnelle prenant la forme d'un logo de l'équipe, obtenue par une modification du flot de conception afin de contraindre le placement des cellules standards. Les cellules, rendues en blanc brillant, forment le circuit, tandis que les points jaunes tout autour représentent le champ électrostatique après placement.

5.4. Encore plus loin ?

L'ouverture matérielle n'est qu'une facette de cette transformation. Ces évolutions rappellent que l'informatique repose sur une pile d'abstractions, depuis les langages de haut niveau jusqu'au silicium. Les problématiques du traitement du signal audio traversent toute cette pile : précision numérique, latence, consommation énergétique, structures de calcul spécialisées. Nous avons démontré qu'avec des infrastructures logicielles comme MLIR (*Multi-Level Intermediate Representation*)¹⁰, il devient possible d'exprimer ces transformations multi-niveaux de manière systématique (Ledoux, Cochard, and Dinechin, 2025 ; Cochard et al., 2025). On peut ainsi imaginer un flot de compilation unifié allant d'un

¹⁰ <https://mlir.llvm.org/>



langage audio haut niveau jusqu'à la génération automatique d'un *bitstream* FPGA ou d'un GDSII ASIC. La frugalité matérielle devient alors non seulement un objectif technique mais un véritable paradigme reliant informatique, microélectronique et création sonore.

6. Conclusion

Les évolutions des technologies de circuits intégrés sont confrontées à deux changements majeurs : la fin de la loi de Moore et l'arrivée de microcontrôleurs très puissants dans le monde de l'embarqué. Ces évolutions ouvrent des opportunités d'innovation dans le champ des systèmes embarqués audio : utilisation d'architectures exotiques, augmentation de la puissance de calcul disponible, etc. Nous avons présenté l'outil SyFaLa permettant de compiler des programmes Faust sur FPGA, mais il est important de comprendre que de nombreuses autres initiatives vont voir le jour pour aller vers une démocratisation des systèmes embarqués audio.

D'un autre côté, les concepts du logiciel libre s'étendent à des domaines connexes comme la conception ouverte d'objets matériels (électroniques, mécaniques, etc.). Nous avons présenté les évolutions des outils libres dans le domaine de la conception électronique, qui permettent d'envisager aussi leur utilisation par un public plus large (artistes, designers, etc.). L'appropriation de ces nouvelles technologies par les communautés artistiques reste un défi aujourd'hui.

7. Bibliographie

- Ajayi, T, D Blaauw, TB Chan, CK Cheng, VA Chhabria, DK Choo, M Coltella, et al. 2019. "OpenROAD: Toward a Self-Driving, Open-Source Digital Layout Implementation Tool Chain." *Proc. GOMACTECH*, 1105–10.
- Ajayi, Tutu, Vidya A Chhabria, Mateus Fogaça, Soheil Hashemi, Abdelrahman Hosny, Andrew B Kahng, Minsoo Kim, et al. 2019. "Toward an Open-Source Digital Flow: First Learnings from the Openroad Project." In *Proceedings of the 56th Annual Design Automation Conference 2019*, 1–4.
- Bell, Michael. 2023. "Tt-Audio-Pmod - TinyTapeout Audio Pmod Interface." <https://github.com/MichaelBell/tt-audio-pmod>.
- Benz, Thomas, Paul Scheffler, Nils Wistoff, Philippe Sauter, Beat Muheim, and Luca Benini. 2025. "ArtistIC: Rendering Artistic Layouts for Integrated Circuits." *arXiv Preprint*. <https://arxiv.org/abs/2502.02626>.
- Bilbao, Stefan. 2009. *Numerical Sound Synthesis: Finite Difference Schemes and Simulation in Musical Acoustics*. Wiley Publishing.
- Cochard, Pierre, Luc Forget, Florent de Dinechin, and Louis Ledoux. 2025. "Towards Multi-Level Arithmetic Optimizations." In *EuroLLVM Developers' Meeting*. <https://hal.science/hal-05063466>.
- Cochard, Pierre, Maxime Popoff, Antoine Fraboulet, Tanguy Risset, Stéphane Letz, and Romain Michon. 2023. "A Programmable Linux-Based FPGA Platform for Audio DSP." In *Proceedings of the 2023 Sound and Music Computing Conference (SMC-23)*, 110–16. Bresin, R., & Falkenberg, K.



- Cochard, Pierre, Maxime Popoff, Romain Michon, and Tanguy Risset. 2024. "Programming FPGA Platforms for Real-Time Audio Signal Processing In C++." In *2024 Sound and Music Computing Conference (SMC-24)*. Porto, Portugal.
- Cochard, Pierre, Jurek Weber, Romain Michon, Tanguy Risset, and Stéphane Letz. 2024. "Ethernet Real-Time Audio Transmission to FPGA." In *2024 IEEE 5th International Symposium on the Internet of Sounds (IS2)*. <https://doi.org/10.1109/IS262782.2024.10704104>.
- Edwards, Timothy. 2023. "TinyTapeout – 36 Sine Synth." https://tinytapeout.com/chips/tt1hp25a/tt_um_rte_sine_synth.
- Emeraude Team. 2025. "Syfala Repository." <https://github.com/inria-meraude/syfala>.
- Everest, Wallie. 2023. "TinyTapeout 3 - ChipTune." <https://tinytapeout.com/runs/tt03/001>.
- Gerzon, Michael J. 1973. "Periphony: With-Height Sound Reproduction." *Journal of The Audio Engineering Society* 21: 2–10.
- Google, and SkyWater. 2020. "The SkyWater Open Source PDK." <https://www.skywatertechnology.com/google-partners-with-skywater-and-efabless-to-enable-open-source-manufacturing-of-custom-asics/>.
- Grame-CNCM. 2025. "Faust Website." <https://faust.grame.fr/>.
- Knoll, Thorsten. 2024. "TinyTapeout 6 – AudioChip_V2." https://tinytapeout.com/chips/tt06/tt_um_thorkn_audiochip_v2.
- Lavington, Simon Hugh. 1980. *Early British Computers: The Story of Vintage Computers and the People Who Built Them*. Art in Context. Manchester University Press.
- Ledoux, Louis, and Marc Casas. 2022. "A Generator of Numerically-Tailored and High-Throughput Accelerators for Batched GEMMs." In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 1–10. <https://doi.org/10.1109/FCCM53951.2022.9786164>.
- . 2023. "An Open-Source Framework for Efficient Numerically-Tailored Computations." In *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*, 19–26. <https://doi.org/10.1109/FPL60245.2023.00011>.
- Ledoux, Louis, Pierre Cochard, and Florent de Dinechin. 2025. "Towards Optimized Arithmetic Circuits with MLIR." *WiPiEC Journal - Works in Progress in Embedded Computing Journal* 11 (1): 4–4. <https://doi.org/10.64552/wipiec.v11i1.90>.
- Letz, Stéphane, Yann Orlarey, and Dominique Fober. 2018. "FAUST Domain Specific Audio DSP Language Compiled to WebAssembly." In *Companion Proceedings of the The Web Conference 2018*, 701–9. WWW '18. Republic; Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee. <https://doi.org/10.1145/3184558.3185970>.
- Meng, Yeo Kheng. 2023. "TinyTapeout 4 - PWM Audio." https://tinytapeout.com/runs/tt04/tt_um_yeokm1_pwm_audio.
- Michon, Romain, Joseph Bizien, Maxime Popoff, and Tanguy Risset. 2024. "Making Frugal Spatial Audio Systems Using Field- Programmable Gate Arrays." In *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME-23)*, 54–59. Zenodo. <https://doi.org/10.5281/zenodo.11189100>.
- Michon, Romain, Yann Orlarey, Stéphane Letz, Dominique Fober, and Dirk Roosenburg. 2020. "Embedded Real-Time Audio Signal Processing with Faust." In *International Faust Conference (IFC-20)*.



- Moore, Gordon. 2006. “Cramming More Components onto Integrated Circuits, Reprinted from Electronics, Volume 38, Number 8, April 19, 1965, Pp.114 Ff.” *Solid-State Circuits Newsletter, IEEE* 11 (October): 33–35. <https://doi.org/10.1109/N-SSC.2006.4785860>.
- Nelson, Philip A. 1994. “Active Control of Acoustic Fields and the Reproduction of Sound.” *Journal of Sound and Vibration* 177 (4): 447–77.
- Orlarey, Yann, Dominique Fober, and Stéphane Letz. 2004. “Syntactical and Semantical Aspects of Faust.” *Soft Computing* 8 (9): 623–32. <https://doi.org/10.1007/s00500-004-0388-1>.
- Popoff, Maxime, Romain Michon, and Tanguy Risset. 2024. “Enabling Affordable and Scalable Audio Spatialization With Multichannel Audio Expansion Boards for FPGA.” In *2024 Sound and Music Computing Conference (SMC-24)*. Porto, Portugal.
- Popoff, Maxime, Romain Michon, Tanguy Risset, Pierre Cochard, Stéphane Letz, Yann Orlarey, and Florent De Dinechin. 2023. “Audio DSP to FPGA Compilation.” In *Proceedings of the 2023 IEEE 34th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 31–32. <https://doi.org/10.1109/ASAP57973.2023.00018>.
- Popoff, Maxime, Romain Michon, Tanguy Risset, Pierre Cochard, Stéphane Letz, Yann Orlarey, and Florent de Dinechin. 2023. “Audio DSP to FPGA Compilation: The Syfala Toolchain Approach.” RR-9507. Univ Lyon, INSA Lyon, Inria, CITI, Grame, Emeraude. <https://inria.hal.science/hal-04099135>.
- Popoff, Maxime, Romain Michon, Tanguy Risset, Yann Orlarey, and Stéphane Letz. 2022. “Towards an FPGA-Based Compilation Flow for Ultra-Low Latency Audio Signal Processing.” In *Proceedings of the 2022 Sound and Music Computing Conference (SMC-22)*, 555–62.
- Quiédeville, Benjamin, Romain Michon, Tanguy Risset, and Stéphane Letz. 2025. “The Space Bar: An Embedded WFS Sound System.” In *Proceedings of the 32th Journées d’Informatique Musicale*. Lyon, France: GRAME and Inria. <https://hal.science/hal-05102322>.
- Ren, Shihong, Stéphane Letz, Yann Orlarey, Romain Michon, Dominique Fober, Michel Buffa, and Jerome Lebrun. 2020. “Using Faust DSL to Develop Custom, Sample Accurate DSP Code and Audio Plugins for the Web Browser.” *Journal of the Audio Engineering Society* 68 (10): 703–16.
- Scherzer, Maximilian. 2024. “TinyTapeout 8 – Sine Wave Synthesizer (SWS).” https://tinytapeout.com/chips/tt08/tt_um_08_sws.
- Shalan, Mohamed, and Tim Edwards. 2020. “Building OpenLANE: A 130nm OpenROAD-Based Tapeout- Proven Flow : Invited Paper.” In *2020 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 1–6.
- Shirriff, Ken. 2021. “Reverse-Engineering the Yamaha DX7 Synthesizer’s Sound Chip from Die Photos.” <https://www.righo.com/2021/11/reverse-engineering-yamaha-dx7.html>.
- Smith, Julius Orion. 2007. *Introduction to Digital Filters: With Audio Applications*. Vol. 2. W3K Publishing.
- Vannoy, Trevor, Tyler Davis, Connor Dack, Dustin Sobrero, and Ross Snider. 2019. “An Open Audio Processing Platform Using SoC FPGAs and Model-Based Development.” In *Audio Engineering Society Convention 147*. Audio Engineering Society.
- Wolfram, Stephen. 2002. *A New Kind of Science*. Wolfram Media. <https://www.wolframscience.com>.



-
- Ziemer, Tim. 2018. “Wave Field Synthesis.” In *Springer Handbook of Systematic Musicology*, 329–47. Berlin, Heidelberg: Springer Berlin Heidelberg.
- Zioma, Renaldas. 2023. “TinyTapeout 5 – SN76489 Programmable Sound Generator.” <https://github.com/rejunity/tt05-psg-sn76489>.
- . 2024a. “Drop Project – an Audio-Visual Demo on an Open-Source ASIC.” In *Proceedings of ORConf 2024, FOSSi Foundation*. Gothenburg, Sweden. <https://fossi-foundation.org/orconf/2024>.
- . 2024b. “TinyTapeout 5 - AY-3-8913 Programmable Sound Generator.” <https://tinytapeout.com/runs/tt05/165>.
- Zotter, Franz, and Matthias Frank. 2019. *Ambisonics: A Practical 3D Audio Theory for Recording, Studio Production, Sound Reinforcement, and Virtual Reality*. <https://doi.org/10.1007/978-3-030-17207-7>.