Faust2FPGA for Ultra-Low Audio Latency: Preliminary work in the Syfala project

Tanguy Risset, Romain Michon, Yann Orlarey, Stéphane Letz, Gero Müller, Adeyemi Gbadamosi

Proceedings of the 2nd International Faust Conference (IFC-20), Maison des Sciences de l'Homme Paris Nord, France, December 1-2, 2020

FAUST2FPGA FOR ULTRA-LOW AUDIO LATENCY: PRELIMINARY WORK IN THE SYFALA PROJECT

Tanguy Risset,[a] Romain Michon,[b,c] Yann Orlarey,[b] Stéphane Letz,[b]
Gero Müller,[a] Adeyemi Gbadamosi,[a] Luc Forget,[a] and Florent de Dinechin[a]

[a]CITI, Insa, Lyon, France
[b]GRAME, Lyon, France
[c]CCRMA, Stanford University, USA
tanguy.risset@insa-lyon.fr *

ABSTRACT

FPGAs are increasingly present in every field of computer science. The generalization of High Level Synthesis (HLS) improves the productivity of the FPGA programmer. However, even with HLS, FPGA configuration requires advanced engineering. This paper investigates the use of FPGA in the context of very low latency (less than 500μs) audio digital signal processing. We propose a methodology to *compile* FAUST programs on FPGA platforms towards ultra-low latency. We expose the challenges it raises and report about the first steps of a future `faust2FPGA` compiler.

1. INTRODUCTION

Embedded systems for audio and multimedia are increasingly used in the arts and culture (e.g., interactive systems, musical instruments, virtual and augmented reality, artistic creation tools, musical composition and performance, etc.). However, programming them can be out of reach of artists, creators, or non-specialized engineers. In parallel with the emergence of the "maker culture,"[1] progress have been made to make these types of systems more accessible, bringing more flexibility in digital approaches to artistic creation. Domain Specific programming Languages (DSL) such as FAUST [1] facilitated the implementation of real-time audio Digital Signal Processing (DSP) algorithms.

However, many limitations remain, especially for real-time applications where latency plays a crucial role (e.g., efficient active control of sound where audio processing should be faster than the propagation of sound [2], digital musical instruments playability [3], digital audio effects, etc.). While latency can be potentially reduced on "standard" computing platforms such as personal computers based on a CPU (Central Processing Unit), going below the "one millisecond threshold" is usually impossible due to buffering.

FPGAs (Field Programmable Gate Arrays) can help solve this problem as well as most of the limitations of traditional computing platforms used for musical and artistic applications. These chips are known for their high computational capabilities [4, 5] and very low-latency performances [6]. They also provide a large number of GPIOs (General Purpose Inputs and Outputs) which can be exploited to implement modern real-time multi-channel processing algorithms (e.g. sound fields capture using lots of digital microphones [7], active sound control over a large spatial region [8], etc.).

* This work was supported by FIL `https://fil.cnrs.fr/` and Inria ADT program. Ousmane Touat has also contributed to it during his internships.

[1] `https://makerfaire.com/maker-movement` (All URLs were verified on Oct. 23, 2019).

But FPGAs remain extremely complex to program, even with state-of-the-art high-level tools, making them largely inaccessible to musicians, digital artists and makers communities. FPGAs are configured/programmed using a Hardware Description Language (HDL) such as VHDL or Verilog. The learning curve and the electrical engineering skills required to master these types of environments make them out of reach of the real-time audio DSP community. Solutions exist to program FPGAs at a higher level (i.e., LabVIEW, Vivado HLS, etc.), but none of them is specifically dedicated nor adapted to real-time audio DSP.

This paper describes the work-in-progress occurring in the Syfala[2] project [9]. The goal of Syfala is to design an FPGA-based platform for multichannel ultra-low-latency audio Digital Signal Processing programmable at a high-level with FAUST and usable for various applications ranging from sound synthesis and processing to active sound control and artificial sound field/room acoustics. The questions addressed in this paper are the following: What would be an ideal `faust2FPGA` compiler? What is achievable and what is not? Can we achieve such a project with reasonable engineering efforts?

Section 2 presents in more details the context and the problem we're addressing. Section 3 presents the challenges to overcome in order to obtain a real compiler from FAUST to FPGA. Section 4 presents our prototype system: using high level synthesis to map FAUST programs on Xilinx FPGA. Then we briefly present some performance results in section 5.

2. CONTEXT AND PROPOSAL

Low-latency and real-time audio implied sustained efforts since the advent of the first digital audio systems. However, *ultra-low latency* has recently opened doors to new applications. We refer to *ultra-low latency* when the delay between the input signal sampling and the output response is less than 500μs. If sampled at 48kHz, a delay of one sample sample costs approximately 20μs, hence we are seeking for systems with a reaction time standing between 1 and 25 samples.

Ultra-Low Audio Latency for What? This type of system has a wide range of applications in multiple domains. Music technology is in high demand for low latency because it helps increasing the playability of musical instruments on stage. In that context, the computational power of the FPGA can be exploited to run

[2] Syfala (for "Synthétiseur Faible Latence sur FPGA") is a local project between GRAME-CNCM and Citi-lab funded by the Fédération Informatique de Lyon (FIL).

complex algorithms (e.g., physics-based models of musical instruments, modal reverbs [10], etc.) that are too costly to run on a traditional platform (i.e., laptop, etc.).

Another field of application for the platform is active control of acoustical spaces (e.g., noise cancellation in rooms, car passenger compartment, etc.) and virtual room acoustics (e.g., to apply the acoustical properties of a space to another, etc.). Sound field rendering systems are in high demand for low audio latency (i.e., to beat acoustical waves traveling at the speed of sound in the air) and high computational power (i.e., to implement Finite Difference Time Domain [11]). While FPGAs have been used in the past for this type of applications [12], there is a lack of a high level tool to program and implement these types of algorithms. Similarly, FPGAs should allow to run in real-time room acoustics simulation algorithms such as modal reverbs [13] in the time domain, which is hardly possible on regular computers. Such experiments have been attempted in the past at the Center for Computer Research in Music and Acoustics (CCRMA)[3] at Stanford University. For instance, the acoustics of the Hagia Sophia cathedral in Istanbul has been recreated in Stanford's Bing Concert Hall for a series of concerts centered around the idea of "archeoacoustics" [10].

Finally, ultra-low audio latency on FPGA has direct industrial applications outside of the field of audio (e.g., aircraft jet engine vibration control, etc.) [7].

Barriers to Low Latency Real-time audio can be reached by means of dedicated real-time operating systems or bare-metal implementations. Bare-metal – i.e., without OS – audio applications have a considerably lower latency and offer new opportunities [14]. High performance DSPs of GPUs can be used to improve bandwidth (i.e., computed number of samples per second), but latency is directly linked to the size of the buffer used by the audio driver.

In general, computation is performed by audio drivers using a buffer of a given number of samples (between 8 and 4096 samples, typically). This is because memory accesses have to be grouped in order to take advantage of the cache hierarchy present on all computers. This is a fundamental limitation of the use of software computers for ultra-low latency audio processing and has been studied for a long time [15, 16].

Existing FPGA-Based Audio Systems FPGAs are known for their high computational capabilities [4, 5] and very low-latency performances [6]. They also provide a large number of GPIOs (General Purpose Inputs and Outputs) which can be exploited to implement modern real-time multi-channel processing algorithms (e.g., sound fields capture using a very large number of digital microphones [7, 17], active sound control over a large spatial region [8], various signal processors for audio [18, 19]). The resulting FPGA designs are seldomly open source.

There are currently only a few examples of professional FPGA-based real-time audio DSP systems (i.e., Antelope Audio,[4] Korora Audio,[5] Focusrite Pro,[6] etc.) and in these applications, FPGAs are dedicated to a specific task, limiting creativity and flexibility.

Recently Vaca et al. presented an open audio processing platform based on the Zybo board [20] for a dedicated purpose such as, for instance, collecting analog frequencies of a musical instrument. They had very similar engineering issues as presented in this paper, but they did not target ultra-low latency ($250ms$) and they did not use HLS to have a fast compilation flow.

HLS Comes of Age Programming FPGA is usually done with a hardware description language (VHDL or Verilog). Developing a VHDL IP[7] is extremely time consuming. Hence, FPGA programmers have two possibilities: re-using existing IPs and assemble them to compose a circuit solving their problem (as proposed by LABVIEW[8]), or use High-Level Synthesis to *compile* a VHDL specification from a higher-level description.

High Level Synthesis (HLS) [21] has been referred to for decades as *the* means to enable fast and safe circuit design for programmers. However, the design space offered to a hardware designer is so huge that no automatic tool is able to capture all the constraints and come up with the *optimal* solution. Many HLS tools have been proposed (e.g. Pico [22], CatapultC [23], Gaut [24], to cite a few) dedicated to specific target application domains. Most of the existing tools start from a high-level representation which is based on a programming language (C, C++, or Python) which is *instrumented* using pragmas to guide the HLS process.

Using HLS today still requires very specific skills [25] to write a source description that is correctly processed by the HLS tools, but we believe that this technology has reached a certain maturity and can now be foreseen as a valuable tool for audio designer. For `faust2FPGA`, we have been focusing on `vivado_hls` ,[9] developed by Xilinx for Xilinx platforms.

FAUST as Source Audio Language FAUST is a DSL for real-time audio signal processing primarily developed at GRAME-CNCM and by a worldwide community. FAUST is based on a compiler translating DSP specifications written in FAUST into a wide range of lower-level languages (e.g., C, C++, Rust, Java, WASM, LLVM bitcode, etc.). Thanks to its "architecture" system, generated DSP objects can be embedded into template programs (wrappers) used to turn a FAUST program into a specific ready-to-use object (e.g., standalone, plug-in, smartphone app, Web page, etc.).

As a data-flow language, FAUST could be naturally translated into a hardware description language simply by translating each *box* of its graphical representation. Moreover the high flexibility of its compiler enables the generation of C++ code dedicated to a specific HLS tool. Hence, the `faust2FPGA` compilation flow is split into several successive steps: FAUST compilation to C++ (using the FAUST compiler), HLS flow using `vivado_hls`, FPGA synthesis, implementation, and bitstream generation using `vivado` (if the FPGA targeted is from Xilinx).

3. FPGA COMPILATION FOR AUDIO CHALLENGES

This section reviews the challenges that an audio programmer might face when generating FPGA configuration. The required

[3] `https://ccmra.stanford.edu`

[4] `https://en.antelopeaudio.com`

[5] `https://www.kororaaudio.com`

[6] `https://pro.focusrite.com/designing-the-ultimate-interface`

[7] IP stands for Intellectual Property, it is the common denomination for *hardware library*, i.e., a circuit design that can be re-used as for instance a software library

[8] `https://www.ni.com/fr-fr/shop/labview.html`

[9] `https://www.xilinx.com/HLS`

technical skills span a wide area from, of course, audio signal processing to embedded system programming, through advanced compilation techniques and hardware design.

Audio Chip Interface The first difficulty is to choose an FPGA with the right features and an audio chip (i.e., audio codec) with specific requirements compatible with the chosen FPGA. It should be large enough to handle large programs and cheap enough to make the final device affordable. Also, as we will see in the following sections, the audio processing algorithm type has consequences on the choice of the FPGA.

Selecting the appropriate audio chip for the type of application that we target is also an important step. Audio chips are used to sample analog signals and perform analog and digital filtering to limit aliasing effects. Many audio chips are not designed for ultra-low latency and might induce a latency of several hundred micro-seconds (a feature called *group delay* in audio chip data sheets [16]). This latency is usually dependent on the sampling frequency. Other audio chips (more expensive) such as the Analog Devices ADAU177[10] limit latency to $50\mu s$.

Another more technical solution is to get rid of the audio chip and to "design" an ADC and a DAC on the FPGA itself. This would allow a much higher sampling rate and open many design opportunities but it would also imply some electronic work to implement the missing analog filters.

While there exists many FPGA boards that can be used for real-time audio applications, designing a custom board is often the only solution as existing boards almost never fit specific needs. Once a development board has been selected (i.e., with an audio chip accessible directly from the IO pins of the FPGA), it must be configured for a use where the FPGA (and not the processor next to it) controls the audio chip.

The FPGA communicates with the audio chip by a serial protocol. Many audio chip use the I2C and I2S protocols. Open-source VHDL code can be found for these protocols,[11] but their integration and debugging process can be tedious.

The I2C IP is used to configure the audio chip with the correct parameters (e.g., sampling rate, bit-width of samples, etc.), while the I2S is used for serializing the 24 bit stereo audio samples on the serial link used to communicate with the audio chip. Once these IPs are integrated, one should be able to run a design such as the one represented on Fig. 1, where a FAUST IP (i.e., performing signal processing on 24 bit wide audio samples) is connected to the I2S IP, itself connected to the audio chips (`sd_tx` and `sd_rx` are serial data transmitted to SSM2603 audio chip).

Note that this part of the process only concerns classical engineering, but the debugging process of this first design can be quite long because simple audio streams such as DC offsets (i.e., constant values) will be suppressed by the audio chip. This phase implies the use of a logic analyzer for debugging I2C and I2S protocols using some additional GPIOs of the FPGA.

Hardware and Software Control Interface Once the I2C/I2S interface is working, a mean of interacting with the FAUST IP must be selected. Assuming that we are processing stereo signals with 24 bit-width such as what is shown on Fig. 1, choosing other parameters – say for instance: compute on *two* parallel stereo signals – will lead to other HDL designs which cannot be parameterized. In that case, the whole design process would have to be re-engineered because it would contain additional IO ports.

It seems acceptable to recompile a hardware design when the number of audio channel is changing, but FAUST programs have other parameters such as the ones created by UI elements (e.g., `hslider`, `button`, etc.). Hence the question is: how to allow the user to modify them and how to provide a *generic* design allowing DSP parameters to vary?

There are many possibilities for interacting with the design on modern FPGAs. Controllers – basically integers or floating points whose values can be changed from the outside, but which is not bound by the same low latency constraints – can be stored in many places: block RAMs, dedicated IP registers, in external memory, etc. Modification of these values can be controlled by an embedded OS sitting next to the FPGA or using GPIOs connected to sensors (e.g., potentiometers, etc.) through an ADC requiring some kind of I2C implementation, etc. Again, the "default" design will probably have a fixed maximal number of such physical controllers (the design of Fig. 1 does not implement any controller interface).

Fixed Point vs. Floating Point Since FAUST usually targets CPUs, it uses floating point to encode audio samples and computations on these samples. However, floating point arithmetic on FPGA is terribly more expensive than using fixed point. Integrated DSPs can perform fixed point operation at low cost and there are dedicated tools [26] that implement all operators used in FAUST programs in a highly optimized manner on FPGAs. This is even more true when the operator can be specialized, for instance if one of its operands is constant.

However, it is not sufficient to simply change all the types in the C code from float to a fixed point type. Care should be taken to preserve enough dynamic range to avoid over/underflow of intermediate results. Failing doing so could result in a dramatical degradation of computation precision [27, 28]. This is clearly a problem that requires more than simple engineering to be tackled.

Usually 24 bits for input and output are sufficient, but how wide intermediate computations should be is not always clear. As this is not currently included in the FAUST semantics, this has to be either fixed with automatic tools [29], or cleverly engineered and simulated to find a correct (but yet cheap) encoding of fixed points integers.

Having a better understanding of output precision requirements could also improve the overall architecture.

Storing Samples: External Memory of Block RAMs This is probably the most challenging problem that will have to be solved by the audio FPGA compiler.

Figure 2 presents a simple echo FAUST program and a excerpt of the generated C++ code using floating point representation for computations (i.e., the interface of the IP that will be generated). Since 64536 float samples are needed (which corresponds to more than 2Mbits), it cannot be done using FPGA Block RAM (60 Block RAM of 36 Kbits in the FPGA of the Zybo board). A bus must be used to access external memory that is usually available on the FPGA board to address this type of problem (Xilinx will use an AXI bus, and connect to SDRAM through system memory controller for instance).

[10] `https://www.analog.com/en/products/adau1777.html`

[11] I2C and I2S from Digikey for instance `https://www.digikey.com/eewiki/pages/viewpage.action?pageId=10125324`

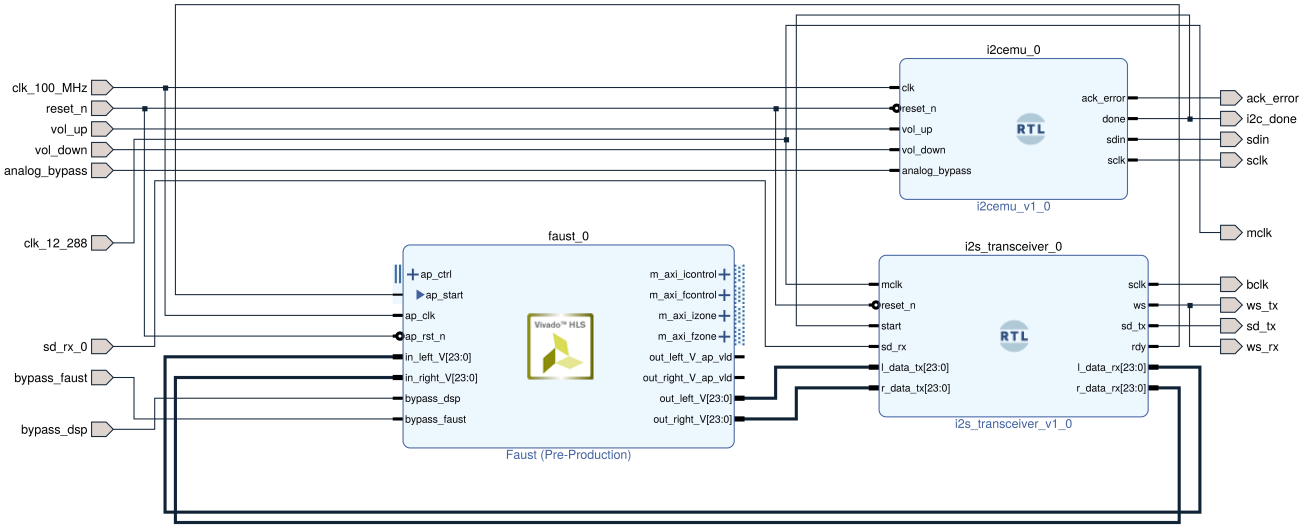

Figure 1: Snapshot of the Vivado block design obtained by connecting FAUST IP, I2C, and I2S IPs

```
//File echo.dsp
myecho = par(i, 2, echo(delay, fback))
with {
echo(d,f) = + ~ (@(d) : *(f));
delay = hslider("delay", 22500, 1, 30000, 1) - 1;
fback = hslider("feedback", 0.7, 0, 0.99, 0.01);
};

process = myecho;
```

```
//file echo.cpp generated from echo.dsp
[...]
// Control arrays
static int icontrol[1];
static FAUSTFLOAT fcontrol[1];
static int izone[1];
static FAUSTFLOAT fzone[65536];

void faust_v3(ap_int<24> in_left, ap_int<24> in_right,
              ap_int<24> *out_left, ap_int<24> *out_right,
      FAUSTFLOAT *fzone)
{
#pragma HLS interface m_axi port=fzone
[...]
```

Figure 2: A simple echo FAUST program with two controllers (top), and an excerpt of the generated code for input to HLS using one AXI bus for the `fzone` array. The memory needed to store the samples is more than 2Mbits, it cannot be done using FPGA Block RAM.

On Figure 3, the same `echo.dsp` has been compiled in a different manner: the four arrays (`fzone`, `izone`, `fControl`, and `iControl` have been each assigned to a different AXI bus (Zync IP proposes up to 4 AXI accesses). Of course, this is not a good choice, because all arrays except `fzone` are limited to 1 element (see Fig. 2). But looking at the generated code (function `computemydsp`), we can see that many memory accesses are issued by this simple program (actually 14 read accesses and 3 write accesses). These memory accesses can be quite time-consuming (more than 100 cycles). Even if the FPGA is clocked at 100MHz (10ns clock period), sequential access – there are four buses but only one memory – should last more than 100*14*10ns=14μs which is almost the time between two samples (20μs). In practice, the C++ code represented in Fig. 3 will not be synthesized correctly by HLS. This illustrates how a bad assignment of variables to memory buffer can be disastrous, and it reminds us that HLS is still a very manual process: we have to find the right way to assign variables to memory and indicate to the HLS tool how memory accesses should be processed.

This memory problem requires thorough attention, there are FPGA chips that include more embedded memory on the FPGA, but one has to bear in mind that, in FAUST, delays can vary from one sample to another. For instance, the `delay` in Fig. 2 can itself be a signal, hence it is very difficult in that case to pipeline memory accesses and prefetch samples from the memory to hide memory access latency.

4. THE SYFALA PROJECT: FAUST COMPILATION ON THE ZYBO

The goal of the Syfala project was to implement a first proof of concept of a VHDL compilation of a FAUST program. We chose to target a Xilinx platform given our experience with Xilinx tools. Xilinx provides an HLS tool (`vivado_hls`) that can generate IP designs easily importable in `vivado` block designs. The chosen FPGA platform was the Zybo-Z7 10[12] – successor of the ZedBoard – that contains a Xilinx Zynq-7000 family FPGA (xc7z010clg400-1) and the recent SSM2603 Audio Codec [30].

[12] `https://reference.digilentinc.com/reference/programmable-logic/zybo-z7/`

```
[...]
void computemydsp(mydsp* dsp, FAUSTFLOAT* inputs,
       FAUSTFLOAT* outputs, int* iControl,
       FAUSTFLOAT* fControl, int* iZone,
       FAUSTFLOAT* fZone) {
   fZone[0+(iZone[0] & 32767)] =
     ((float)inputs[0] +
     (fControl[0] * fZone[0+((iZone[0] - iControl[0]) & 32767)]));
   outputs[0] = (FAUSTFLOAT)fZone[0+((iZone[0] - 0) & 32767)];
   fZone[32768+(iZone[0] & 32767)] = ((float)inputs[1] +
     (fControl[0] * fZone[32768+((iZone[0] - iControl[0]) & 32767)]));
   outputs[1] = (FAUSTFLOAT)fZone[32768+((iZone[0] - 0) & 32767)];
   iZone[0] = (iZone[0] + 1);
}

[...]

void faust(ap_int<24> in_left, ap_int<24> in_right,
           ap_int<24> *out_left,ap_int<24> *out_right,
   int *icontrol, FAUSTFLOAT *fcontrol, int *izone,
   FAUSTFLOAT *fzone, bool bypass_dsp, bool bypass_faust)
{
#pragma HLS interface m_axi port=icontrol
#pragma HLS interface m_axi port=fcontrol
#pragma HLS interface m_axi port=izone
#pragma HLS interface m_axi port=fzone
[...]
```

Figure 3: Example of the same `echo.dsp` file compiled with 4 AXI accesses, the `computemydsp` function is the one to be synthesized by HLS, it performs the computations on streams, one sample at a time.

4.1. Possible Design Flows

The first choice to make is the compilation flow. There are several possibilities involving more or less engineering. They are illustrated in Fig. 4.

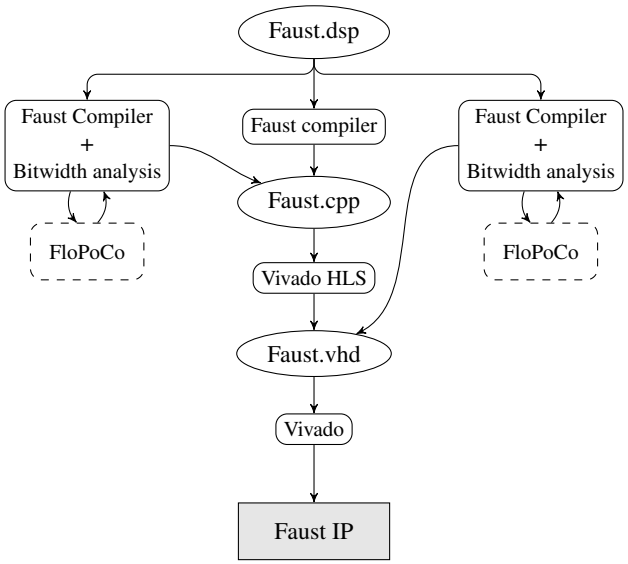

Figure 4: The different possible compilation flows for a `faust2VHDL` compiler, round boxes are files (i.e., format), squared boxes are tools. Dashed boxes are tools outside from the compilation flow used to tune the architecture.

The easiest path is the central one: the FAUST compiler is tuned to output C++ code that fits into `vivado_hls` tools. The type used for the samples in this flow is `float`. Then, the generated VHDL code is packed as an IP and integrated into a block design, connected to I2C/I2S and AXI buses if needed, as shown on Fig. 5. As mentioned before, an important optimization of this design flow is to choose a memory access organization adapted to the compiled FAUST program.

The left flow of Fig. 5 adds the fixed-point optimization. Here, the type chosen for the computation is fixed point of any width. First, an analysis of the FAUST data-flow graph should indicate what the required bit-width for each signal is, then the C++ code is generated accordingly and sent to `vivado_hls` using dedicated arithmetic operators. The `FloPoCo` tool [26] will help to perform such analysis and to optimise filter design.

Finally (right flow), one could think of a direct compiler from FAUST to VHDL. Indeed, the FAUST flow graph representation is close to a structural representation of the computations. The main difficulty here lies in the inference of memory accesses. If simple FIFOs (First In First Out) can be used, it will be easy, but we must keep in mind that delays can be controlled by signals, and change for each sample.

4.2. First Syfala Compilation Flow Choices

The first stable Syfala compilation flow follows the schematics of Figure 6. The choices that have been made are the following:

- Implement a *one sample* flag in the FAUST compiler (`-os`) that generates a `computemydsp` function of the `faust.cpp` file that computes only one sample. It implies that the FPGA signal processing treatment is not pipelined among the audio samples.
- Have a fixed interface of the `faust` function that will be synthesized by `vivado_hls`. This interface is shown in the architecture file `FPGA.cpp` on Fig. 7, with the following conventions:
  - Stereo input and output (i.e., `in_left`, `in_right`, `out_left`, `out_right`) are 24 bits wide signed integers encoding floating point values between -1 and 1, as explained later, which are to be sent and received from the I2S transceiver, which itself interfaces with the audio chip.
  - There are four memory zones identified: the `i/fcontrol` are used to store control parameters, the `{i/f}zones` are used to store samples (`fzone`) or sample index (`izone`). The FAUST compiler ensures coherent access in these different memory zones in the generated C++ code.
  - In the first version of the compiler, for simplicity, these four memory arrays are stored on the external RAM of the Zybo board and hence need an AXI bus interface IP to be accessed from the FPGA.

The `FPGA.cpp` file is the FAUST *architecture file* corresponding to the FPGA target architecture (currently only Xilinx architectures are supported by Syfala). As mentioned above, four pragmas are used to indicate that the four memory zones are all placed in external memory and accessed with a different AXI bus.

The `controlmydsp` and `computemydsp` functions are executed at each sample. The underlying assumption it that the generated IP will be triggered by a `start` signal (i.e., a basic hand shake protocol), indicating that the next sample is ready to be processed.

The `scaleFactor` value (i.e., 8388608.0f) is exactly 2^{23}. The input/output of the `faust` function are arrays of type `ap_int<24>`, i.e., signed integer of 24 bits (`vivado` fixed point library), they are interpreted as decimal part of signed samples between -1 and 1.

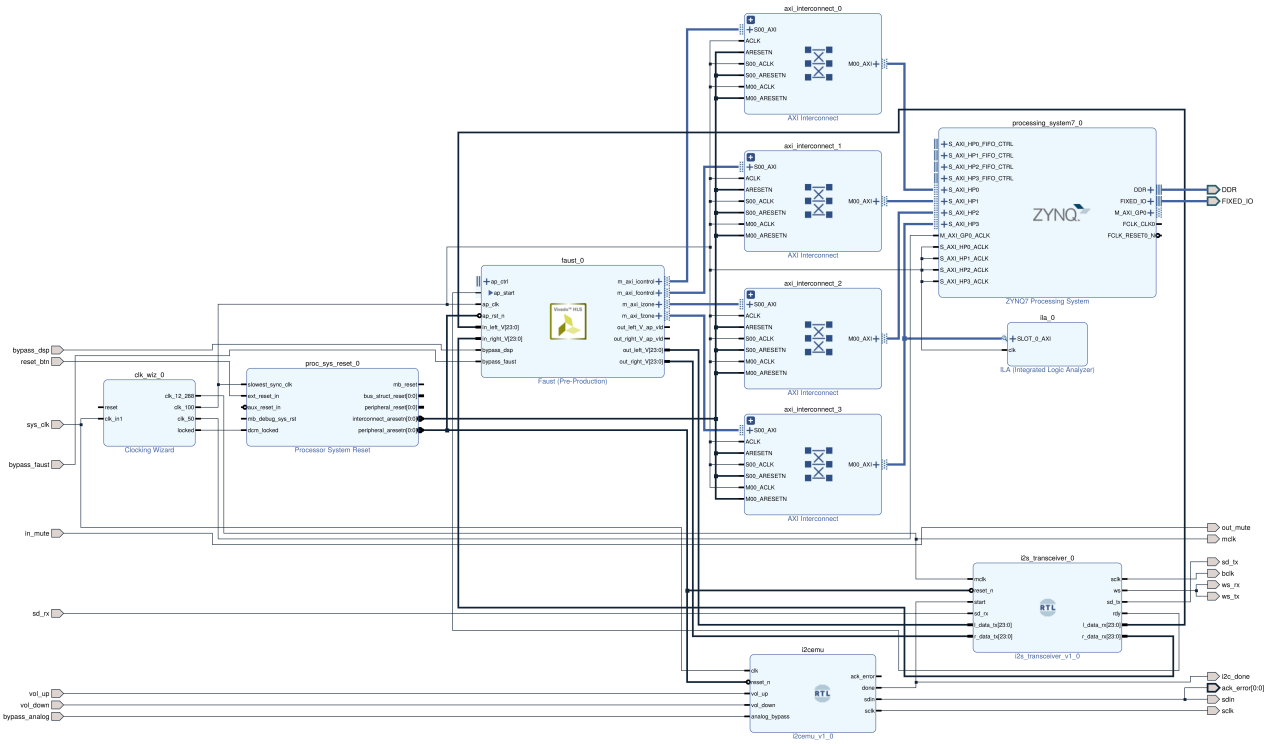

Figure 5: The complete block design obtained after FAUST IP integration (4 AXI buses used here).

The following table shows the correspondence between the floating point values output by the `computemydsp` function and the corresponding sample input to the I2S transceiver:

FAUST `output` Float sample value (a)	value truncated for `24 bits` (b)	value stored in `out_left` (c)
0.12345678123456	0.1234567	$c = a \times 2^{23} = 1035630$
-0.12345678123456	-0.1234567	$c = a \times 2^{23} = -1035630$

4.3. The I2C/I2S IPs

The I2C and I2S VHDL IPs have been implemented from Scott Larson's IP proposed at DigiKey eewiki.[13] The `i2cemu` IP is the first one activated after reboot. It configures the registers of the `SSM2603` chip with the values indicated in the SSM2603 datasheet (see [30], p20). It also proposes an output volume adaptation connected to two buttons of the Zybo board. `i2cemu` IP implements the I2C protocol for configuring the SSM2603 register, then waits for a given period of time (75ms), and activates the start signal sent to the IPs `faust` and `i2s_transceiver`.

The `i2s_transceiver` is the one that actually transmits the bits between the FPGA and the audio codec. The protocol used in our design is the one illustrated in Fig. 8: the 24 bits are serially transmitted along the `bclk` clock (see also [30]). The `ws` signal chooses between left and right channel. The `i2s_transceiver` is then connected to the `faust` IP as explained in Fig. 7.

[13] `https://www.digikey.com/eewiki/pages/viewpage.action?pageId=10125324`

4.4. Time, Clocks, and the Ordering of Ticks in the Syfala System

It is important to understand the origin and value of the different clocks in the system. The generation of the different clocks is simplified by the use of the `Clocking Wizard` IP, which itself inputs the FPGA system clock (`sys_clk`) and outputs the required clocks.

- **FPGA system Clock: 125Mhz** The *internal* FPGA clock that triggers every register of the FPGA depends on the complexity of the design (i.e., the complexity of the longest combinatorial path), it is called `sys_clk` in the Vivado block design. We usually impose this clock to be 125Mhz (i.e., setting a 8ns clock when creating `vivado` and `vivado_hls` projects). If `vivado` fails in synthesizing a design that can be clocked at that speed, it will issue an error message, however it should be easy to change this clock to another value as all other clocks are generated independently of this one.
- **Audio codec internal Master Clock:** $mclk = 256 \times f_s$ The clock regulating the SSM2603 should be a multiple of the sampling frequency. In the I2C configuration [30], we have configured the chip to run with $f_s = 48kHz$ sampling rate, hence:

$$mclk = 256 \times f_s = 256 \times 48kHz = 12.288MHz$$

In our design, this clock is generated with the `clocking Wizard` IP and transmitted to both `i2s_transceiver` and `ssm2603` codec to ports name `mclk`).

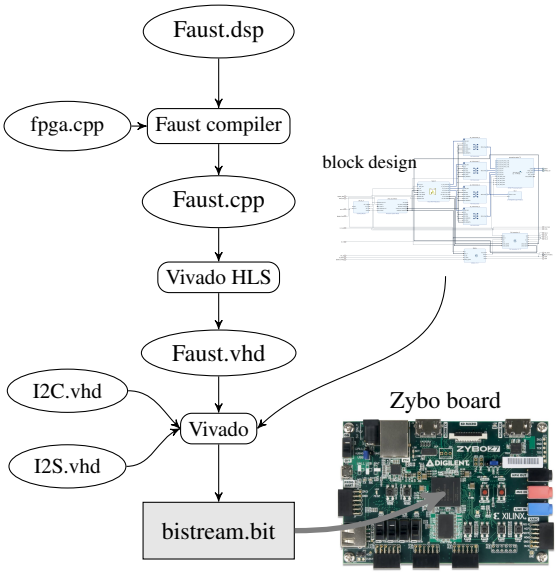

Figure 6: Compilation flow of the first Syfala version `syfala-v1.0` targeting the Zybo board with `vivado`.

- **Vivado IP's clocks** The AXI bus and the Zynq processing system require a 100Mhz clock. The Zynq processing system requires also a 50Mhz clock.
- **The `i2cemu` clocks** The clock used in `i2cemu` to clock the I2C communication with the codec is independent from the clocks used in I2S. We have hardwired (with a generic parameter of the VHDL component) the main I2C clock to 400Mhz. This clock is also called `sclk` in the Vivado block design, but it is not to be confused with the `sclk` of I2S. If the FPGA system clock (125MHz) is changed, the VHDL code f `i2cemu` should be modified accordingly.
- **The `i2s_transceiver` clocks** The I2S transceiver is using two more clocks: the `sclk` clock, sometimes called `bclk` (*bit clock* because it is clocking each bit as illustrated in Figure 8) and the `ws` clock (word select) which selects the left or right channel (illustrated as `reclrc/pblrc` in Fig. 8).

  There is a fixed ratio between these two clocks and the `mclk` mentioned above: `mclk/sclk=4` (i.e., `mclk` is 4 times faster `sclk`) and `sclk/ws=64`. Again, this is hard-coded in `i2s_transceiver` generic VHDL parameters. Hence, one `ws` period is $T_{ws} = 4 \times 64 \times T_{mclk} = 256 \times T_{mclk} = T_{audio} = \frac{1}{48kHz} = 20.83\mu s$.

5. PERFORMANCE AND CONCLUSION

Designs generated by the current version of the Syfala tool-chain are quite complex and they are not completely operational yet: the echo example make sound some samples are lost because of the memory access latency through AXI busses. The only valid FAUST programs produced with the Syfala V4 design flow, are obtained by using only Block RAMs (i.e., no AXI bus) for memory accesses. Table 1 shows the usage of FPGA resources for each IP. Version 1 of Syfala, which used four AXI buses for the four I/O arrays of

```
#define FAUSTFLOAT float
typedef struct {
        FAUSTFLOAT fHslider0;
        FAUSTFLOAT fHslider1;
        int fSampleRate;
} mydsp;
[...]
static char initialized = 0;
static mydsp DSP;

void faust(ap_int<24> in_left, ap_int<24> in_right, ap_int<24> *out_left,
           ap_int<24> *out_right, int *icontrol, FAUSTFLOAT *fcontrol,
           int *izone,FAUSTFLOAT *fzone, bool bypass_dsp, bool bypass_faust)
{
#pragma HLS interface m_axi port=icontrol
#pragma HLS interface m_axi port=fcontrol
#pragma HLS interface m_axi port=izone
#pragma HLS interface m_axi port=fzone

        if (initialized == 0) {
                initmydsp(&DSP, SAMPLE_RATE, izone, fzone);
                initialized = 1;
        }

        // Update control
        controlmydsp(&DSP, icontrol, fcontrol, izone, fzone);

        // Allocate 'inputs' and 'outputs' for 'compute' method
        FAUSTFLOAT inputs[FAUST_INPUTS], outputs[FAUST_OUTPUTS];

        const float scaleFactor = 8388608.0f;

        // Prepare inputs for 'compute' method
#if FAUST_INPUTS > 0
        inputs[0] =  in_left.to_float() / scaleFactor;
#endif
#if FAUST_INPUTS > 1
        inputs[1] =  in_right.to_float() / scaleFactor;
#endif

        computemydsp(&DSP, inputs, outputs, icontrol, fcontrol, izone, fzone);

        // Copy produced outputs
        *out_left = ap_int<24>(outputs[0] * scaleFactor);
#if FAUST_OUTPUTS > 1
        *out_right = ap_int<24>(outputs[1] * scaleFactor);
#else
        *out_right = ap_int<24>(outputs[0] * scaleFactor);
#endif
}
```

Figure 7: The *architecture* file used by FAUST on Syfala v1.0

the IP was the first prototype to run a realistic echo (although some samples were lost as we noticed afterwards). We simplified it in Syfala V3 to only have one AXI access. These versions are not allowing to change controller values yet. Table 1 is shown here to illustrate the fact that the design of the integration of the FAUST IP has an important impact on the complexity of the resulting circuit. Indeed both designs V1 and V3 are using exactly the same FAUST initial program and their resulting complexity are quite different.

On Table 1, it can be seen that the FAUST IP in V1 approximately uses 20% of the LUT ($100 * 3477/17600 = 19.75$) and 4 Block RAMs while the V3 uses only 12% of the LUTs and 1 block RAM. The (constant) complexity of the I2C and I2S transceivers (approximately 2% of the LUTs) is a useful information too. The V1 design size is quite important (about 45% of the FPGA resources), because of the used 4 AXI bus and of the use of an integrated logic analyser (ILA) for debugging purposes. This complexity will be probably easily reduced.

What is more problematic is the fact that despite the simple FAUST program (echo program of Fig. 2), the `Faust_v3` IP accounts for about 10% of the FPGA of the Zybo board. There are two reasons for that:

- First, the float type used for computations is using a lot of resources. We should move to a fixed point version.
- Many unnecessary computations are implemented in hardware, such as the initialization of signals and data. They should be implemented in software on the ARM processor, because there is no latency pressure on them.

Syfala runs on Linux, compilations of these designs have been completely scripted [9] and are activated using `make`. Vivado

Name	Slice LUTs 17600	Slice Registers 35200	Block RAM Tile 60	DSPs 80
Syfala V1 (Four AXI)				
main_wrapper	7584	10742	10	13
axi_interconnect_0	175	262	0	0
axi_interconnect_1	175	262	0	0
axi_interconnect_2	175	262	0	0
axi_interconnect_3	175	262	0	0
faust_v1	3477	4657	4	13
i2cemu	182	178	0	0
i2s_transceiver	180	212	0	0
ila	2543	3883	6	0
proc_sys_reset	17	37	0	0
Syfala V3 (One AXI)				
main_wrapper	2703	3443	1	13
axi_interconnect	175	262	0	0
faust_v3	2119	2742	1	13
i2cemu	213	190	0	0
i2s_transceiver	180	212	0	0
proc_sys_reset	17	37	0	0

Table 1: FPGA resources usage for Syfala V1 (with four AXI and one integrated logic analyzer – ila – for debug purpose) and Syfala V3 (one AXI) for the simple echo FAUST program (Fig. 2). The `Faust_v3` IP still accounts for about 10% of the FPGA of the Zybo board.

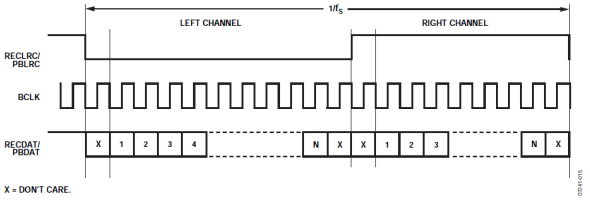

Figure 8: I2S mode used as protocol between `i2s_transceiver` and the audio codec SSM2603 (from [30]) for 24 bits samples ($N = 24$).

allows us to export a script for every tool, however the manual changes on these scripts to make them portable to a git repository are quite complex. The compilation time of these design took between 10 and 20 minutes using Vivado 19.1.

The latency achieved between inputs and outputs of audio samples of this first generated design is about $820\mu s$. As mentioned earlier, $800\mu s$ are introduced by the audio chip, the FAUST IP itself has a delay of approximately $20\mu s$ (1 sample which is the minimum delay as I/O of the IP are synchronised).

We are now studying solutions to improve design complexity using fixed point arithmetic and ways to interact with the controllers.

6. ACKNOWLEDGMENTS

This work was supported by the FIL `https://fil.cnrs.fr/` and the Inria ADT program. Ousmane Touat has also contributed to this project during an internship.

7. REFERENCES

[1] Yann Orlarey, Stéphane Letz, and Dominique Fober, *New Computational Paradigms for Computer Music*, chapter "Faust: an Efficient Functional Approach to DSP Programming", Delatour, Paris, France, 2009.

[2] Stephen Elliott, *Signal Processing for Active Control*, Elsevier, 2000.

[3] Nelson Lago and Fabio Kon, "The quest for low latency," in *Proceedings of the International Computer Music Conference (ICMC-04)*, Miami, USA, 2004.

[4] Jiwon Choi, Myeongsu Kang, Yongmin Kim, Cheol-Hong Kim, and Jong-Myon Kim, "Design space exploration in many-core processors for sound synthesis of plucked string instruments," *Journal of Parallel and Distributed Computing*, vol. 73, no. 11, pp. 1506–1522, 2013.

[5] Florian Pfeifle and Rolf Bader, "Real-time finite difference physical models of musical instruments on a field programmable gate array (fpga)," in *Proceedings of the 15th International Conference on Digital Audio Effects (DAFx-12)*, York, UK, 2012.

[6] Math Verstraelen, Jan Kuper, and Gerard J.M. Smit, "Declaratively programmable ultra low-latency audio effects processing on fpga," in *Proceedings of the 17th International Conference on Digital Audio Effects (DAFx-14)*, Erlangen, Germany, 2014.

[7] Edouard Salze, Emmanuel Jondeau, Antonio Pereira, Simon L. Prigent, and Christophe Bailly, "A new MEMS microphone array for the wavenumber analysis of wall-pressure fluctuations: Application to the modal investigation of a ducted low-Mach number stage," in *Proceedings of the 25th AIAA/CEAS Aeroacoustics Conference*, Delft, Netherlands, 2019.

[8] Jihui Zhang, Thushara D. Abhayapala, Wen Zhang, Prasanga N. Samarasinghe, and Shouda Jiang, "Active noise control over space: A wave domain approach," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 26, no. 4, pp. 774–786, April 2018.

[9] "Syfala gitlab," https://gitlab.inria.fr/risset/syfala.

[10] Jonathan S Abel, Wieslaw Woszczyk, Doyuen Ko, Scott Levine, Jonathan Hong, Travis Skare, Michael J Wilson, Sean Coffin, and Fernando Lopez-Lezcano, "Recreation of the acoustics of hagia sophia in Stanford's Bing concert hall

for the concert performance and recording of cappella romana," in *Proceedings of the International Symposium on Room Acoustics*, Toronto, Canada, 2013.

[11] Stefan Bilbao, *Numerical Sound Synthesis: Finite Difference Schemes and Simulation in Musical Acoustics*, John Wiley and Sons, Chichester, UK, 2009.

[12] Yiyu Tan and Toshiyuki Imamura, "An fpga-based accelerator for sound field rendering," in *Proceedings of the 22nd International Conference on Digital Audio Effects (DAFx-19)*, Birmingham, UK, 2019.

[13] Jonathan S Abel, "Method and system for artificial reverberation using modal decomposition," Apr. 16 2019, US Patent App. 10/262,645.

[14] Romain Michon, Yann Orlarey, Stéphane Letz, and Dominique Fober, "Real time audio digital signal processing with faust and the teensy," in *Proceedings of the Sound and Music Computing Conference (SMC-19), Malaga, Spain*, 2019.

[15] Michael Lester and Jon Boley, "The effects of latency on live sound monitoring," 2007.

[16] Yonghao Wang, Ryan Stables, and Joshua Reiss, "Audio latency measurement for desktop operating systems with onboard soundcards," in *Audio Engineering Society Convention 128*. Audio Engineering Society, 2010.

[17] Dimitris Theodoropoulos, Catalin Bogdan Ciobanu, and Georgi Kuzmanov, "Wave field synthesis for 3d audio: Architectural prospectives," in *Proceedings of the 6th ACM Conference on Computing Frontiers*, New York, NY, USA, 2009, CF '09, p. 127–136, Association for Computing Machinery.

[18] Mihalis Psarakis, Aggelos Pikrakis, and Giannis Dendrinos, "Fpga-based acceleration for tracking audio effects in movies," 04 2012, pp. 85–92.

[19] Jingbo Zhang, Ganggang Ning, and Shufang Zhang, "Design of audio signal processing and display system based on soc," in *2015 4th International Conference on Computer Science and Network Technology (ICCSNT)*. IEEE, 2015, vol. 1, pp. 824–828.

[20] K. Vaca, M. M. Jefferies, and X. Yang, "An open audio processing platform with zync fpga," in *2019 IEEE International Symposium on Measurement and Control in Robotics (ISMCR)*, Sep. 2019, pp. D1–2–1–D1–2–6.

[21] R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, Oct 2016.

[22] Robert Schreiber, Shail Aditya, Scott A. Mahlke, Vinod Kathail, B. Ramakrishna Rau, Darren C. Cronquist, and Mukund Sivaraman, "PICO-NPA: high-level synthesis of nonprogrammable hardware accelerators," *VLSI Signal Processing*, vol. 31, no. 2, pp. 127–142, 2002.

[23] Thomas Bollaert, *Catapult Synthesis: A Practical Introduction to Interactive C Synthesis*, pp. 29–52, Springer Netherlands, Dordrecht, 2008.

[24] Farhat Thabet, Philippe Coussy, Dominique Heller, and Eric Martin, "Exploration and rapid prototyping of DSP applications using systemc behavioral simulation and high-level synthesis," *Signal Processing Systems*, vol. 56, no. 2-3, pp. 167–186, 2009.

[25] R. Kastner, J. Matai, and S. Neuendorffer, "Parallel Programming for FPGAs," *ArXiv e-prints*, May 2018.

[26] Florent de Dinechin and Bogdan Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, July 2011.

[27] Anastasia Volkova, Matei Istoan, Florent de Dinechin, and Thibault Hilaire, "Towards hardware IIR filters computing just right: Direct form I case study," *IEEE Transactions on Computers*, vol. 68, no. 4, Apr. 2019.

[28] Karthick Parashar, Daniel Menard, and Olivier Sentieys, "A polynomial time algorithm for solving the word-length optimization problem," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, San Diego, United States, Nov. 2013.

[29] O Sentieys, D Menard, and N Simon, "Id. fix: an eda tool for fixed-point refinement of embedded systems," Design Automation and Test in Europe (DATE) 2014 University booth.

[30] Analog Devices, "Low power audio codec ssm2603 data sheet," https://www.analog.com/en/products/ssm2603.html.