Towards an FPGA-Based Compilation Flow for Ultra-Low Latency Audio Signal Processing

Maxime Popoff,[a] Romain Michon,[b] Tanguy Risset,[a] Yann Orlarey,[c] and Stéphane Letz[c]
[a]Univ Lyon, INSA Lyon, Inria, CITI, EA3720, 69621 Villeurbanne, France
[b]Univ Lyon, Inria, INSA Lyon, CITI, EA3720, 69621 Villeurbanne, France
[c]Univ Lyon, GRAME-CNCM, INSA Lyon, Inria, CITI, EA3720, 69621 Villeurbanne, France
maxime.popoff@insa-lyon.fr

ABSTRACT

Field Programmable Gate Arrays (FPGAs) have been increasingly used in recent years for real-time audio Digital Signal Processing (DSP) applications. They provide unparalleled audio latency and processing power performances. They can target extremely high audio sampling rates and their large number of General Purpose Inputs and Outputs (GPIOs) make them particularly adapted to the development of large scale systems with an extended number of analog audio inputs and outputs. On the other hand, programming them is extremely complex and out of reach to non-specialized engineers as well as to most people in the audio community. In this paper, we introduce a comprehensive FPGA-based environment for real-time audio DSP programmable at a high level with the FAUST programming language. Our system reaches unequaled latency performances (11 µs round-trip) and can be easily controlled using both a software graphical user interface and a dedicated hardware controller taking the form of a sister board for our FPGA board. The implementation of the system is described in details and its performances are evaluated. Directions for future work and potential applications are presented as well.

1. INTRODUCTION

Audio Digital Signal Processing (DSP) has been widely studied and implemented on a wide range of computer architectures: Von Neuman CPUs, multi-cores, GPUs, dedicated circuits, FPGAs, [1] etc., with throughput (or computing power) as the main target, in most cases. However, achieving low audio latency on such systems is an important objective too that is often limited by hardware or the use of an Operating System (OS). For instance, on conventional computers (i.e., "PCs"), samples must be grouped in buffers in order to "hide" from the audio codec chip [2] potential latency induced by interruptions, etc.

The only solution to fully get rid of buffering on digital audio systems – and hence reach the lowest possible latency – is to use a dedicated circuit (i.e., ASIC [3] or FPGA) directly connected to an audio codec. The benefits of using such systems for real-time audio DSP expand far beyond the scope of audio latency by: (i) potentially providing more computational power and throughput, (ii) allowing for the use of very high audio sampling rates (>10MHz), (iii) providing a large number of audio inputs and outputs. However, using FPGAs makes the implementation of DSP algorithms extremely complicated because hardware design [4] is a long and tedious process. In this context, the use of FPGA or VLSI [5] design tools – which are high-added-value proprietary environments – cannot be avoided. However, there has been a trend in recent years towards making them more accessible, which primarily translated into the rise of High Level Synthesis (HLS) for non-professional hardware designers.

In this paper, we propose to simplify the implementation of audio DSP programs on FPGA platforms by providing a fully automated *compilation flow* based on a high-level audio DSP language: FAUST [1]. The FAUST compiler is used in conjunction with HLS tools to program the system down-to hardware. This compilation flow is optimized for audio latency and is able to reach unparalleled figures: 11 µs from analog input to analog output. We demonstrate the use of our tool-chain on the Xilinx Zybo Z7 board connected to various audio codecs. We also introduce a hardware control interface taking the form of a sister board for the Zybo Z7 to facilitate the control of audio DSP on the FPGA.

The actual added values of our work lie in (i) the use of HLS and (ii) solving the most important technical obstacles related to the automatic translation of an audio DSP language to an FPGA, namely: software control of the DSP, initialization of constants, and minimization of external memory accesses.

[1] *Field Programmable Gate Array*

[2] In this paper, "audio codec" always refer to a hardware component providing analog outputs and inputs (audio ADC/DAC): not an audio compression algorithm.

[3] *Application Specific Integrated Circuit*

[4] In the world of FPGA, the term "hardware" can be ambiguous and actually refers to what is happening on an FPGA chip. Hence, "hardware design" here is equivalent to "FPGA programming." This terminology will be used throughout the paper.

[5] *Very Large Scale Integration*

2. STATE OF THE ART

2.1 FPGA and Audio DSP

Audio on FPGA has been studied for a long time with initial contributions in the 2000s [2–4] and more recently [5–7], to only mention a few. Most of these works consist in manually designing a hardware/software implementation of a particular audio process. FPGA design was so complex that these implementations were actually published as research papers. In 2014, Verstraelen proposed an innovative dedicated architecture synthesized [6] on FPGA that could be programmed from data-flow graphs of audio applications [8], but this is not an active project anymore.

In the industry, an increasing number of products are using FPGAs for real-time audio signal processing tasks such as the Novation Summit [7] or some Antelope Audio [8] products (non-exhaustive list).

FPGA design methodologies are evolving quickly and it is now possible to derive hardware description from a high-level specification. High Level Synthesis is now proposed by all FPGA vendors and can be seen as a way to increase hardware designers productivity. It consists in *compiling* a high-level language (usually restricted syntax of C, C++, or Python) down-to hardware. Examples of such tools are Xilinx's `Vitis_HLS`, Mentor Graphic's `Catapult-C` or Intel/Altera's `HLS compiler`. The term *compilation* is a bit reductive because HLS transforms a sequential program into a functionally equivalent circuit containing many parallel computations, opening the door to much more computational power.

IP-based design is another approach proposed by MathWorks HDL coder [9] or Xilinx XSG. [10] A recent study at Lund University [9] compared MathWork HDL coder with vendor tools, but did not extend the comparison to HLS tools. This evolution opened the way to a new category of papers on FPGA audio starting from higher level specifications of audio DSP, greatly facilitating the overall engineering process.

The work of Vannoy et al. [10, 11] is one of the most advanced in this category. They use MathWork HDL coder and an Intel Cyclone V FPGA. Their complete design is available on GitHub and they provide many additional facilities (i.e., Linux sound card driver, web client for control interface, etc.). An interesting work on blind sailing was proposed by Singhani and Morrow [12]. They mapped "by hand" a Head-Related Transfer Function (HRTF) on an FPGA to achieve low latency performances (with a latency below 5ms as an aim). LowLAG [13] and [14] are other examples of these approaches.

Our FAUST2FPGA compilation flow is implemented in the SyFaLa toolchain [11] [15]. Our system is the only one using HLS and providing an actual *compilation* process. Other works all rely on existing Matlab/Simulink blocks or FPGA Vendors blocks for their design. Additionally, we are the only ones providing support for the automatic use of external DDR memory when Block Rams are not sufficient. Finally, we are also proposing a hardware interface to control audio DSP on the FPGA (see §4).

2.2 Low Audio Latency

The music and audio technology community has been focusing on audio latency for a very long time. Wang's PhD thesis [16] provides a good review of the different potential sources of delay in digital audio systems. For instance, ADC/DAC, [12] operating systems, and audio networking are usually identified as the main contributors to audio latency.

When using FPGAs, the only potential source of latency comes from the ADC/DAC because no operating system is involved to process the sample stream. Most modern audio codecs on the market are based on $\Delta\Sigma$ ADC/DACs. $\Delta\Sigma$ DACs imply the use of filters for signal reconstruction which are the main source of latency on this kind DACs. The codecs that we decided to use are presented in §4.1 and are specifically optimized for latency, mostly by using advanced reconstruction filter designs implying a minimal number of taps (as opposed to more basic codecs that typically use high order digital filters). There is a clear lack of accurate delay reporting mechanisms for $\Delta\Sigma$ ADC/DAC so the effective delay of a given system usually needs to be evaluated experimentally as shown in §4.1.

Among the aforementioned works, the smallest announced latency on FPGA processing, from analog input to analog output is $180\,\mu s$ [10]. In §5.1, we demonstrate how to further reduce latency by a factor of 10.

3. AUDIO DSP TO FPGA COMPILATION PRINCIPLES

Audio DSP programs have a specific structure which can be better expressed in *Domain Specific Languages* (DSLs) such as CSound [17], FAUST, or PureData [18]. The efficient compilation of audio DSP programs to hardware must take this particular structure into account. This paper presents a compilation flow of FAUST programs, but the basic principles used here can be adapted to other DSLs.

This section gives an overview of the structure of audio DSP programs and how it is handled for efficient compilation to FPGA.

3.1 Audio DSP to FPGA Compiler Overview

Figure 1 presents an overview of the FAUST2FPGA compilation flow. This compilation flow has been tested on the Xilinx Zynq-7010 SoC, [13] which is used on the Zybo Z7 board. [14] A FAUST source program – say a filter-based sine wave oscillator such as the one presented on

[6] The term "synthesis" here refers to the process of turning a program specified in a Hardware Description Language (HDL) into a bitstream (as opposed to "sound synthesis").

[7] `https://novationmusic.com/en/synths/summit` (All URLs in this paper were verified on Jan 25, 2022)

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

[9] `https://www.mathworks.com/products/hdl-coder.html`

[10] `https://www.xilinx.com/products/design-tools/vitis/vitis-model-composer.html`

[11] `https://github.com/inria-emeraude/syfala`

[12] *Analog to Digital Converter // Digital to Analog Converter*

[13] *System on a Chip*

[14] `https://www.xilinx.com/products/boards-and-kits.html`

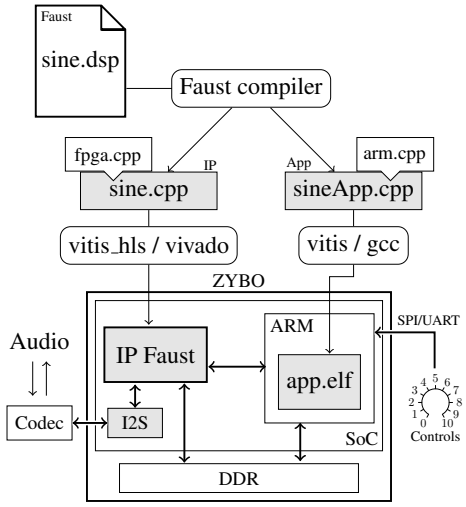

Figure 1. FAUST to FPGA compilation flow, gray boxes are generated during the compilation flow

Fig. 2 (taken from the FAUST libraries) – is compiled to C++, using a specific `fpga.cpp` architecture file and the `-os2` compiler option that was specifically implemented as part of this project (see §3.3). HLS and synthesis tools (`Vitis_HLS` and `Vivado` in our case as we target Xilinx hardware) are used to generate the FAUST IP [15] from the C++ code. The portion of the code associated to control (i.e., the `freq` slider on Fig. 2) is compiled to another C++ program using a specific `arm.cpp` architecture file, to be executed by the ARM processor of the Zynq SoC, which in turn can potentially be interfaced with software or hardware physical controllers (see §4.2). The constants initialization and computations depending on control variables (variables `th`, `c`, and `s` in the `with` region on Fig. 2) are also executed on the ARM processor. The way control is managed in our system is detailed in §4.2. DSP computation – i.e., `nlf2` which implements a second order normalized digital waveguide resonator on Fig. 2 – is executed for each sample on the FPGA by the FAUST IP.

```
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) : !,_  <: _,_;
```

Figure 2. Filter-based sine wave oscillator in FAUST illustrating the compilation process.

The FAUST IP is connected to an I2S IP which interfaces the FPGA with an audio codec. The sample rate and the number of audio Input/Output can be configured before compilation.

[15] *Intellectual Property*

3.2 Audio Objects Topology

3.2.1 General Structure of An Audio Program

In audio DSP programs, it is common to distinguish two different rates: the *audio rate*, which is typically between 20 kHz and 192 kHz, and the *control rate*. The control rate corresponds to the rate at which external controllers are updated (typically between 100 Hz and 1000 Hz). As an example, all computations influenced by the `freq` controller in Fig. 2 can be executed at control rate. Conversely, computations for the `nlf2` function should be executed at audio rate.

Another characteristics of audio programs is that some computations are done only once at initialization time. They correspond to constant computations which cannot be done statically because they depend on values known at execution time, such as the sample rate. Similarly, static arrays are heavily used in audio DSP programs, either for waveform tables or for delay lines (e.g., echos, reverbs, etc.), translating into large memory footprints (often requiring dozens of MB of memory).

3.2.2 Adapting Audio Programs to FPGA Architectures

A wide range of FPGA-based boards (such as the Digilent Zybo Z7 used here) integrate an ARM CPU which can be used in conjunction with the FPGA. In the context of audio DSP, FPGA resources shall only be used when needed. Hence, some portion of a program can be ran on the ARM processor, and the rest can be processed by the FPGA itself. As mentioned before, the use of an FPGA for audio processing is potentially beneficial to (i) performances, (ii) throughput of the audio program, (iii) using extremely high sampling rates, (iv) getting a large number of audio input and output channels, or (v) for latency reasons.

Constants initialization (i.e., waveform tables, delays, etc.) does not need to be computed by the FPGA and can be carried out on the ARM CPU before starting DSP. While the available memory on the FPGA itself (called *Block Rams*) is rather limited (about 1 MB on Zynq-7010 SoCs), external memory (DDR) is huge, but accessing it can be fifty times slower than Block Rams. As a consequence, Block Rams should be used in priority by objects used in current sample computations to avoid starvation due to high memory latency. On the other hand, multiple memory accesses to the same array element (to the same sample, for instance) should be cached on the FPGA. This is also true on a GPP [16] where several accesses to the same sample should be carried out through a scalar temporary variable that can be cached in a register by the compiler.

3.2.3 Automatizing This Process With the Faust Compiler

The FAUST compiler typing system is able to identify computations that can be done at *compilation time*, *initialization time*, *control rate*, and finally *sample rate*. Thanks to that infrastructure, the aforementioned dispatching can be done easily:

- Constant initialization should be dispatched on the ARM CPU, then copied on the FPGA.

[16] *General Purpose Processor*

```
[....]
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]);
}
[....]
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);
}
[....]
```

Figure 3. Excerpt of C++ code generated by the FAUST compiler from the FAUST code presented on Fig. 2 when tuned for the FPGA target.

- Control rate computations should be dispatched on the ARM CPU and the resulting values copied on the FPGA at control rate.
- Other computations, i.e., sample rate computations, should be dispatched on the FPGA.
- Small objects should be stored on Block Rams as much as possible.
- Large arrays should be stored in external DDR. The caching mechanism is already implemented in the code generated by the compiler as shown by the `fTemp` variables on Fig. 3.

3.3 New -os2 Code Generation Option

A new *-os2* compilation option has been added to the FAUST compiler for the C and C++ backends. Fig. 3 shows an excerpt of the C++ code generated from the code on Fig. 2. It allows for a better separation between control rate and sample rate computations as well as for a better control of the DSP memory layout:

- A separate `controlmydsp` function is generated. It contains the code that will be executed at control rate using the new value of all controllers (i.e., buttons, sliders, etc.). This code is then used to update the FAUST IP internal state. The `controlmydsp` function is called on the ARM side (i.e., right side on Fig. 1).
- The `computemydsp` function is used to compute a single sample (see Fig. 3). It does not contain the loop that is generated in the standard version of the FAUST compiler which would normally process a buffer of input/output samples. Hence, the name of the new option: `-os2` stands for *one sample, version 2*. This function is called on the FPGA side.
- Two additional `iControl` and `fControl` arrays are used as parameters to share the control state between the `controlmydsp` and `computemydsp` functions.
- DSP memory is divided between a section kept in the DSP class/structure, and a separated section typically allocated on the DDR using `iZone` and `fZone` arrays.[17] The DDR will typically contain long delay lines. The user can tune an additional environment variable `FAUST_MAX_SIZE` to precisely control how the memory layout is divided, depending on the amount of memory available in Block Rams.
- Two new functions to copy the constants computed on the ARM side (i.e., in `iZone` or `fZone`) on the FPGA are also generated.

As a summary, these concepts are illustrated by the C++ code excerpt in Fig. 3 generated by the FAUST compiler with the `-os2` option from the DSP program presented in Fig. 2. The `controlmydsp` function is executed on the ARM CPU (`app.elf` in Fig. 1), waiving the need for a floating point Sine and Cosine functions on the FPGA. The portion of the code synthesized on the FPGA is contained in the `computemydsp` function. Caching of duplicated memory accesses can be seen via the `ftemp0` and `ftemp1` variables and exploited by the HLS compiler. In this particular example, there are no large arrays so all variables are stored in Block Rams. Note that the same `-os2` option is used on both the FPGA and ARM sides to preserve the structure of generated code, only the architecture files change (`fpga.cpp` and `arm.cpp` on Fig. 1).

3.4 Faust to FPGA in Practice

As mentioned earlier, FAUST2FPGA is implemented as part of the Syfala compiler [15], which currently only runs on Linux. It is freely accessible on the Syfala GitHub.[18] Installing the Xilinx Tools Suite is a requirement. The control application (`app.elf` on Fig. 1) runs *bare metal* (i.e., without an operating system) on the ARM CPU.

The different tools are called using `make`. Xilinx tools are configured with TCL scripts. Compiling a program such as the one on Fig. 2 takes about fifteen minutes on a standard PC. A big chunk of the time is spent on the Vivado logic synthesis.

4. CONTROLLERS & INTERFACING THE FAUST IP TO THE CODEC

The previous section presented how the FAUST IP is generated from the FAUST code. IP interfacing has to be treated with care because it has an important impact on performances and latency. For instance, in our initial attempt [15] we have shown that a 800 µs latency was added by the built-in audio codec (Analog Devices SSM2603) of the Zybo board, which was not optimized for latency. Section 4.1 explains how we have interfaced the FAUST IP with various codecs in order to obtain the smallest possible latency (11 µs from analog input to analog output). Then Section 4.2 presents the interface for controlling the audio DSP in hardware or in software.

[17] iControl/iZone contain integer values, fControl/fZone contain float values.

[18] `https://github.com/inria-emeraude/syfala`

4.1 Audio Codec Interface

In the current version of the compiler, the FAUST IP has a constant latency of one sample. Hence, only two potential sources of latency can be optimized: the audio codec itself and the serial audio interface (I2S) between the codec and the FAUST IP. Audio codecs usually combine an ADC and a DAC, a serial I2S interface, and some analog filters for signal reconstruction (see Fig. 5). Some codecs also include a hardware DSP (an actual hardware component). In our case, it is only used for signal routing and hence is not a source of additional latency.

In the following paragraphs, the I2S clock rate (which is also the sample rate of our system) is noted as $f_s^{I^2S}$, and the ADC/DAC internal clock rate as Ck^{ADC}. [19] Both can be configured but there are some constraints, in particular on their maximum value.

Three different audio codecs were used for our experiments (very few codecs on the market are optimized for latency and the ADAU1777 and the ADAU1787 are the ones providing the best performances that we could find):

- The Analog Devices SSM2603 (the built-in codec of the Zybo board), which is not optimized for low latency. ($\max f_s^{I^2S} = \max Ck^{ADC} = 96\,\text{kHz}$)
- The Analog Devices ADAU1777, which is optimized for low latency. ($\max f_s^{I^2S} = 192\,\text{kHz}$ & $\max Ck^{ADC} = 768\,\text{kHz}$)
- The Analog Devices ADAU1787, which is the fastest audio codec we used and that provides the best performances in terms of latency. ($\max f_s^{I^2S} = \max Ck^{ADC} = 768\,\text{kHz}$)

The ADC/DAC introduces at least a 1 sample delay. Beyond that, audio latency is directly connected to the sampling rate: the higher the sampling rate, the lower the latency. In practice though, the codec actually introduces more than just one sample of delay. Audio codecs refer to this as *group delay* and corresponds to the delay added by the ADC/DAC. Group delay depends on many parameters, including the sampling rate. For the ADAU1787, the best announced group delay is 5 µs (analog input to analog output), obtained at $Ck^{ADC} = f_s^{I^2S} = 768\,\text{kHz}$. We validated this figure by measuring a 5.9 µs latency when bypassing the I2S, which approximately corresponds to a 4 samples delay (see Section 5.1).

However, this latency does not include the serial audio interface delay. The I2S protocol is clocked with a *bit clock* (Bclk) which depends on the sample rate $f_s^{I^2S}$. The relation between $f_s^{I^2S}$ and f^{Bclk} depends on the bit depth of the audio samples: $f^{Bclk} = f_s^{I^2S} \times 2 \times \text{bit depth}$. Hence, for the ADAU1787, the maximum authorized I2S sample rate is higher with audio samples of 16 bits than for 24 or 32 bits.

Increasing the sampling rate $f_s^{I^2S}$ lowers the latency introduced by the I2S. On the other hand, it decreases the available time for the FAUST IP to perform its computation. So increasing the I2S sampling rate must be done carefully. However, the ADC clock rate Ck^{ADC} can be increased without changing the I2S sampling rate, hence many combinations are possible as explained in Section 5.1.

Once the codec parameters have been optimized, the serial audio interface on the SoC side needs to be managed. To ensure that the latency of the I2S interface is as low as possible, we implemented our own I2S transceiver IP. Prior works [13] use the AXI4-Lite interface but our custom VHDL IP provides us a better control over latency. Our implementation is based on an IP from the digikey forum [20] which was extended to enable arbitrary sample bit depths. Another advantage of this custom I2S IP is that it allows us to adjust the number of channels (audio inputs or outputs). We can therefore produce as many I2S channels as we want – all sharing the same bit clock (Bclk) and Word Select signal – minimizing GPIOs usage to two pins per I2S.

The effective measurements of the latency introduced by the codec interface is presented in Section 5.1. Next section presents the interface used to control the FAUST audio DSP.

4.2 Control Interface

The FAUST IP can be controlled using a physical (i.e., knobs, potentiometers, buttons, etc.) or a software (i.e., running on a computer connected to the FPGA board) interface system which were designed as part our project. They both interact with the FAUST IP through the ARM processor of the FPGA board. When the state of a FAUST DSP parameter is updated on the ARM (whether it came from a software or a hardware action), the updated control state is transferred on the FPGA using the ARM s-axi calling the `sendControlToFPGA` function (see §3.2). This function sends all the new controller states to the FPGA taking a *best effort* strategy: the ARM is infinitely looping and sending new controllers values as fast as it can.

4.2.1 Hardware Interface

The hardware interface is based on a PCB board [21] ("sister board/shield") which can be mounted on top the Zybo Z7 (see Fig. 4). This board is generic enough so that a wide range of combinations of controllers (i.e., buttons, rotary potentiometers, faders, etc.) can explored and soldered on it. It is interfaced to the ARM processor via an ADC chip mounted directly on the board and using the SPI protocol. The physical controllers of the board can be bound to DSP parameters directly in the FAUST program using using metadata such as `[knob:1]` or `[switch:1]`. For example, the `freq` slider of Fig. 2 is bound to the first knob of the hardware interface board, etc. A detailed documentation of this system is available on the SyFaLa GitHub.[18]

4.2.2 Software Interface

The software interface can be automatically generated by the FAUST2FPGA compilation chain (see §3). It is a GTK-based GUI directly adapted from the FAUST architectures environment and that is meant to be executed

[19] Here we will consider that the ADC and the DAC have the same sample rate.

[20] `https://forum.digikey.com/t/i2s-transceiver-vhdl/12845`

[21] The source files of the board are available on the GitHub of the project.[18]

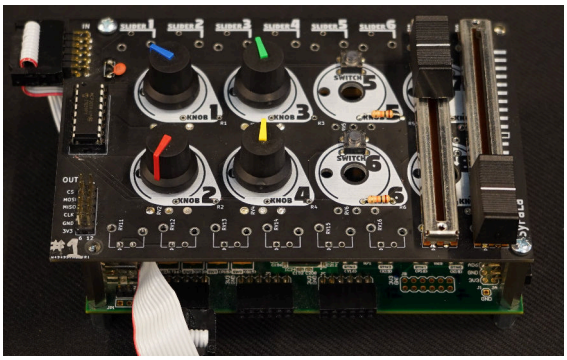

Figure 4. Zybo Z7 with our custom sister board presenting a possible combination of controllers.

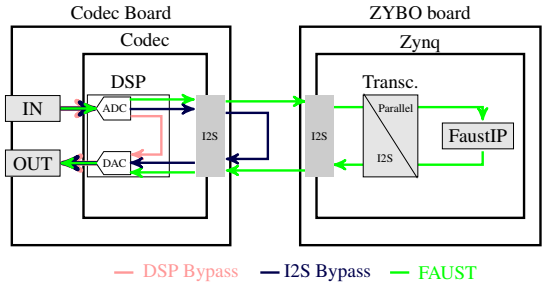

Figure 5. Latency measurements testbench used for the results presented in Table 1. Latency was measured from analog input to analog output using square signals in three different configurations.

on a host computer connected to the Zybo board. Once launched, it communicates with the ARM processor of the Zybo using the UART serial protocol. The corresponding communication classes `UartReceiverUI` and `UartSenderUI` are automatically integrated to the application program and can be used for that purpose. Unlike the hardware controller presented in the previous section, the generated software interface contains all the parameters declared in the FAUST program.

5. RESULTS AND EVALUATION

In this section, we first evaluate the audio latency obtained by our tool-chain. We then measure the impact of our optimizations on memory organization and computations dispatching.

5.1 Latency Performances

Our experimental setup for measuring latency is described in Fig. 5. A 440 Hz square wave signal is sent to a simple FAUST IP implementing a pass through. The latency between the analog input and output of the system is measured by looking at the time difference between zero crossings of the source and of the processed signal. We performed a large number of measurements of this kind. Their standard deviation is always less than one sample.

Three types of measurements were performed for each codec (see Fig. 5) from analog input to analog output:

- a complete pass-through going to the FAUST IP through I2S, as described above (path labeled FAUST in Fig. 5),
- a pass-through going through I2S only (without going through the FAUST IP), this path is labeled "I2S Bypass" in Fig. 5,
- an internal pass-through in the codec (path labeled DSP Bypass), hence not using I2S.

Table 1 shows the latency measurements obtained on various codecs with different sampling rate configurations. The first column indicates the global sampling rate of the system $f_s^{I^2S}$ (the sampling rate used by the I2S interface). The second column indicates the internal sampling rate of the codec: Ck^{ADC} (which therefore can be different from the global sampling rate of the system). As expected, the best latency (11 µs) is obtained with the fastest internal ADC/DAC with an I2S rate of 768 kHz and using 16 bits samples. In the current version of the compiler, the FAUST IP will always add a 1 sample delay. To our knowledge, none of the previous works could achieve such a low latency (i.e., 180 µs was announced in [10] with a codec clock at 768 kHz). Our custom I2S transceiver also adds an additional sample of delay, so the whole SoC (transceiver+FAUST, back and forth) will add 2 samples of delay. Note that the ADAU1787 works at voltage levels up to 1.8V, whereas the Zybo operates at 3.3v. It is therefore necessary to use a level shifter to ensure compatibility between the two boards. The level shifter has to be fast enough to handle the I2S clock (up to 24.576 MHz). We used a LSF0204 level shifter which doesn't induce any significant latency.

	$f_s^{I^2S}$ (kHz)	Ck^{ADC} (kHz)	Latency (μs)	Latency (sample)
SSM2603				
FAUST	48	48	848	41
FAUST	96	96	191	18
ADAU1777				
Bypass DSP	-	768	9.2	-
Bypass I2S	192	768	69.2	13
FAUST	48	96	298.2	14
FAUST	96	96	179.6	17
FAUST	48	768	255.2	12
FAUST	192	768	79.5	15
ADAU1787				
Bypass DSP	-	768	5.9	-
Bypass I2S	192	768	30.3	6
Bypass I2S	768	768	8.4	6
FAUST	48	48	206.2	10
FAUST	96	96	97.2	9
FAUST	48	768	145.6	7
FAUST	192	768	40.8	8
FAUST	384	768	25.4	10
FAUST*	768	768	11.1	8

Table 1. Latency report for different audio codecs. All measurements were made on 24 bits samples, except for the last one (indicated by *) which was made on 16 bits samples. Latency expressed as samples is computed based on the $f_s^{I^2S}$ sampling rate and rounded to the nearest integer.

	LUT (% use)	1 sample comp. time (cycles)	1 sample comp. time (µs)
BRAM synth	66 %	458	3.66 µs
MEM synth	73 %	1477	11.81 µs
BRAM guit. demo	77 %	1922	15.58 µs
MEM guit. demo	84 %	3956	31.64 µs
BRAM lms	64 %	846	7.22 µs
MEM lms	191 %	784	6.27 µs
BRAM tictac	92 %	500	4.00 µs
MEM tictac	95 %	1564	12.51 µs
BRAM comb	34 %	77	0.61 µs
MEM comb	47 %	738	5.90 µs
BRAM violin	109 %	741	6.00 µs
MEM violin	110 %	1955	15.64 µs

Table 2. Memory storage vs. Block Ram storage: comparing performances (Look Up Tables: LUT usage and computation time expressed in cycle of FPGA clock at 125 MHz) for various FAUST programs. Red numbers indicate that the design does not fit on Zybo Z7-10 FPGA. One sample computation time must be less than 20.83 µs at 48kHz (i.e., one sample time).

5.2 Memory Accesses Issues

As mentioned in §3.2, the use of external memory access is often needed for audio programs because FPGA Block Rams do not offer sufficient space for storing delay lines. However, from our observations, a memory access is at least 50 times slower than a Block Ram access. Table 2 shows the impact on latency (i.e., computation time for one sample) of a systematic memory use compared to a systematic use of Block Rams on a set of simple FAUST programs taken from the Amstramgrame website. [22]

These experiments show that using memory during sample computations can possibly increase the computation time of each sample by a factor of 10 (see the `comb` example on Table 2). As explained before, the FAUST compiler has integrated this optimization in the code generated with the `-os2` option. However, in that case, DDR memory should also be used to initialize constants and perform control rate computations (see §3).

5.3 Impact of Initialization

Fig. 7 shows the impact on the FAUST IP size of reporting constant initialization on the ARM processor using `bellN.dsp` as a reference (see Fig 6). Fig. 7 instantiates `N` for integer values between 1 and 30 and compares resources usage (lookup tables and DSPs [23]) for scenarios where initialization is carried on the ARM processor or on the FPGA. It can be seen that no more than 3 resonant filters can be used if FPGA initialization is used, while up to 25 resonant filters can be used if initialization is carried out on the ARM.

These results show that the current version of the `-os2` mode of the FAUST compiler provides adequate optimization of memory accesses for basic FPGA designs. Future work will improve parallelization to reach even better performances.

[22] `https://www.amstramgrame.fr/gramophone/programs/`

[23] DSP here refers to the multipliers integrated in the FPGA.

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

t60 = 30;
pulse = button("gate") : ba.impulsify;

process = pulse : pm.frenchBellModel(N,0,t60,1,2.5);
```

Figure 6. The `bellN.dsp` program used to produce the results presented of Fig. 7. `N` should be replaced by an integer value indicating the number of resonant filters used to provide the bell sound.

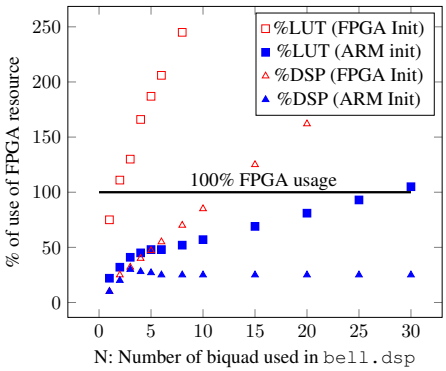

Figure 7. Percentage of Zybo FPGA Look Up Tables (LUTs) and DSPs[23] usage for various sizes (i.e., number of biquads) of the `bell.dsp` FAUST program (see Fig 6), with initialization on the ARM processor (in blue) or on the FPGA (in red).

6. CURRENT AND FUTURE PERSPECTIVES

We are currently working at improving the control of parallelism on the FPGA. The design decisions made for the *one sample* mode have a significant impact because they forbid the pipelining of successive sample computations. Pipelining could possibly solve some performance limitations. On the other hand, it would increase latency, which would be counter-productive. Pipelining successive samples in the FAUST IP would imply to re-think the FAUST IP interface, it could for instance pre-fetch data from memory and overlap more efficiently memory accesses and computations. Along the same lines, we are looking at generating fixed-point code directly from the FAUST compiler which should allow us to generate much more optimized IPs for audio DSP.

The number of audio input/output – 2×2 for our compilation chain – can be increased, but this implies connecting additional codecs and manually adapting the compilation chain (this task remains fairly simple though). Relatively cheap I2S TDM [24] audio codecs ($< \$3$) can now be found on the market. TDM allows us to potentially connect up to eight codecs on the same FPGA GPIO (2 GPIOs if an audio input is needed) opening the possibility to implement systems with an unparalleled number of audio inputs and outputs with potential applications to spatial audio, etc.

Controllers are currently communicating with the Faust IP through the ARM CPU. This implies that the latency of controllers is not necessarily as good as the latency of

[24] *Time Division Multiplexing*

audio streams. This could be improved by connecting the controllers directly to the FPGA.

Finally, the power of FPGAs should allow us to completely get rid of the codec and implement a $\Sigma\Delta$ ADC/DAC directly on the FPGA with simple minimal analog filtering. This would allow us to use much higher audio sampling rate than the ones currently used in our system, opening the door to a wide range of research avenues.

7. CONCLUSION

A large portion of the possibilities offered by FPGAs for real-time audio DSP have yet to be explored. We hope that the work presented in this paper will contribute to making this type of platform more accessible to the audio and music technology research communities. We foresee potential applications for active control (afforded by ultra-low latency) of room acoustics and of musical instruments, spatial audio (with systems potentially offering unparalleled number of audio inputs and outputs), and new approaches to audio DSP with audio sampling rates above 20MHz.

Acknowledgments

This work has been carried out in the context of the FAST ANR project [25] (ANR-20-CE38-0001) funded by the French ANR (Agence National de la Recherche).

8. REFERENCES

[1] Y. Orlarey, S. Letz, and D. Fober, *New Computational Paradigms for Computer Music*. Paris, France: Delatour, 2009, ch. "Faust: an Efficient Functional Approach to DSP Programming".

[2] K. Byun, Y.-S. Kwon, S. Park, and N.-W. Eum, "Digital Audio Effect System-on-a-Chip Based on Embedded DSP Core," *ETRI Journal*, vol. 31, no. 6, pp. 732–740, Dec. 2009.

[3] M. Pfaff, D. Malzner, J. Seifert, J. Traxler, H. Weber, and G. Wiendl, "Implementing digital audio effects using a hardware/software co-design approach," in *10th International Conference on Digital Audio Effects*, 2007, pp. 1–8.

[4] D. Theodoropoulos, G. Kuzmanov, and G. Gaydadjiev, "Multi-Core Platforms for Beamforming and Wave Field Synthesis," *IEEE Transactions on Multimedia*, vol. 13, no. 2, pp. 235–245, Apr. 2011.

[5] J. Zhang, G. Ning, and S. Zhang, "Design of audio signal processing and display system based on SoC," in *2015 4th International Conference on Computer Science and Network Technology (ICCSNT)*. Harbin, China: IEEE, Dec. 2015, pp. 824–828.

[6] C. Dragoi, C. Anghel, C. Stanciu, and C. Paleologu, "Efficient FPGA Implementation of Classic Audio Effects," in *2021 13th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*. Pitesti, Romania: IEEE, Jul. 2021, pp. 1–6.

[7] 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)*. Houston, TX, USA: IEEE, Sep. 2019, pp. D1–2–1–D1–2–6.

[8] M. Verstraelen, J. Kuper, and G. J. Smit, "Declaratively Programmable Ultra Low-Latency Audio Effects Processing on FPGA." in *DAFx*, 2014, pp. 263–270.

[9] E. Pongratz and R. C. John, "Performance Evaluation of MathWorks HDL Coder as a Vendor Independent DFE Generation," 2019, master PhD.

[10] T. Vannoy, T. Davis, C. Dack, D. Sobrero, and R. Snider, "An open audio processing platform using soc fpgas and model-based development," in *Audio Engineering Society Convention 147*. Audio Engineering Society, 2019.

[11] T. C. Vannoy, "Enabling rapid prototyping of audio signal processing systems using system-on-chip field programmable gate arrays," Master PhD, 2020.

[12] A. Singhani and A. Morrow, "Real-Time Spatial 3D Audio Synthesis on FPGAs for Blind Sailing," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Seaside CA USA: ACM, Feb. 2020, pp. 104–110.

[13] Y. E. Esen and I. San, "Low-Latency SoC Design with High-Level Accelerators Specific to Sound Effects," *International Journal of Advances in Engineering and Pure Sciences*, vol. 33, pp. 78 – 87, dec 2021.

[14] L. Merah, P. Lorenz, A. Ali-Pacha, and N. Hadj-Said, "A Guide on Using Xilinx System Generator to Design and Implement Real-Time Audio Effects on FPGA," *International Journal of Future Computer and Communication*, pp. 38–44, Sep. 2021.

[15] T. Risset, R. Michon, Y. Orlarey, S. Letz, G. Müller, and A. Gbadamosi, "faust2fpga for ultra-low audio latency: Preliminary work in the syfala project," in *Proceedings of the International Faust Conference (IFC-20)*, Paris (France), 2020.

[16] Y. Wang, "Low latency audio processing," Ph.D. dissertation, Queen Mary University of London, 2018.

[17] V. Lazzarini, S. Yi, J. Heintz, Ø. Brandtsegg, I. McCurdy *et al.*, *Csound: a sound and music computing system*. Springer, 2016.

[18] M. Puckette, "Pure data: Another integrated computer music environment," *Proceedings of the Second Intercollege Computer Music Concerts*, pp. 37–41, 1996.

[25] `https://fast.grame.fr`