Getting started with the Syfala project: From Faust to FPGA

Tanguy Risset and the Syfala Team

18 février 2020

Table des matières

1 The Syfala objective

The Syfala project (*Synthetiseur Faible Latence pour FPGA*) has started as a FIL project, it will probably continue for a while. This docment explains the technical choices that have been made on the first version of the Syfala toolchain.

The Syfala toolchain is a compilation toolchain of Faust program on FPGA (currently Xilinx Zynq present on Zybo-Z7 10 board). The toolchain itself is explained in another document (*install_toolchain.pdf*), which is reproduced in Annex here (from p 11).

The objective is to compile Faust [1] programs on a FPGA platform with the objective of obtaining a short latency between input and output of the signal.

Audio signal is sampled at (say) 48kHz. Hence one audio sample (i.e. one on each channel, two channels for stereo audio) arrives roughly every 2.083×10^{-5} seconds, hence approximately every $20\mu s$. In general it is considered that the latency (i.e. the time between the input of a sample and its effect on output) cannot go below 1 sample delay (i.e. $20\mu s$). Our initial goal was to achieve a delay of *less than* $100\mu s$.

When performing audio processing with a software system, such as on Linux OS, the sound processing is performed by the audio driver which handles the samples coming from the audio codec. The typical application on these systems will play music files or apply an effect on a stream. Ultra-low latency is usually not a problem on this kind of software, but efficiency is. Efficiency is needed for audio real time (computing at least on sample every $20\mu s$) and for having as few CPU cycles as possible, as audio processing is usually sharing the CPU resources with many other tasks.

For efficiency reason, all audio drivers are using buffers to communicate with the audio codec, it means that one driver activation will compute a bunch of samples, usually 64 or more. If a buffer of 64 samples is used, the latency is at least $1.3ms$ (i.e roughly 64*$20\mu s$), then one have to add the time for interruption handling and audio processing itself. This latency has to be added to the codec latency itself (which usually can be configured, but is not negligible), which makes software solution inaplicable for low latency applications.

Using and FPGA to realize audio processing would take advantage of (i) high computing power (parallelism can be quite high on a FPGA circuit), and more important (ii) low latency (as samples are directly coming from/going to audio codec to/from the FPGA circuit).

Few examples of professional FPGA-based real-time audio DSP systems (i.e., Antelope Audio, [2] Korora Audio, [3] etc.) and in these applications, FPGAs are dedicated to a specific task, limiting creativity and flexibility. Moreover, these designs where realized "by hand" i.e. by register transfer level design (in VHDL or Verilog) of the realized circuits. The idea of the Syfala project is to *compile* an FPGA configuration from a Faust audio processing specification. This is made possible by *High Level synthesis* (HLS) which is a compilation flow that transforms a software code (usually based on C-like syntax) into a HDL representation that can be further compiled with classical FPGA programming suites. The most well known HLS tools are `vivadoHLS` (from `Xilinx`), `C2H` (from `Altera`),

1. https://faust.grame.fr/
2. https://en.antelopeaudio.com
3. https://www.kororaaudio.com

`CatapultC` (from `Mentor Graphics`), but other tools are proposed today to bridge the gap between algorithmic representation and hardware level representation of a computation [4].

This project has been launched within a collaboration between Grame research department and Citi Socrate team because :

1. Faust has been invented in Grame and Grame researchers are the main developers of Faust compiler.
2. Socrate members have skills in FPGA programming, embedded systems and HLS.
3. The FloPoCo [5] tools is developed in Socrate and might be of critical help in designing efficient circuits from Faust.
4. Many new application fields require low latency (from dynamic acoustic adaptation to new digital musical instruments).

2 Syfala compilation flow

To get Syfala compiler working, these are the current `git instruction` to compile a simple echo program using the Syfala v2 using only one AXI access. The complete tool installation is explained in Annex A. In the current version, the branch used in master, each version is deployed in a subdirectory : `v1-four-axi` for Gero's version, `v2-one-axi` for the second Syfala version.

```
git clone https://gitlab.inria.fr/risset/syfala.git mysyfala
cd mysyfala/v2-one-axi
make
```

or if you have installed your ssh key on gitlab :

```
git clone git@gitlab.inria.fr:risset/syfala.git mysyfala
cd mysyfala/v2-one-axi
make
```

The first stable Syfala compilation flow follows the schematics of Figure 2.1

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 the following one :

```
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)
```

with the following conventions :

— Stereo input and output (i.e. `in_left`, `in_right`, `out_left`, `out_right`) are 24 bit wide signed integer between -1 and 1, which are to be send and receive from the i2s transceiver which himself will interface with the audio codec.

4. See https://en.wikipedia.org/wiki/High-level_synthesis for instance
5. http://flopoco.gforge.inria.fr/

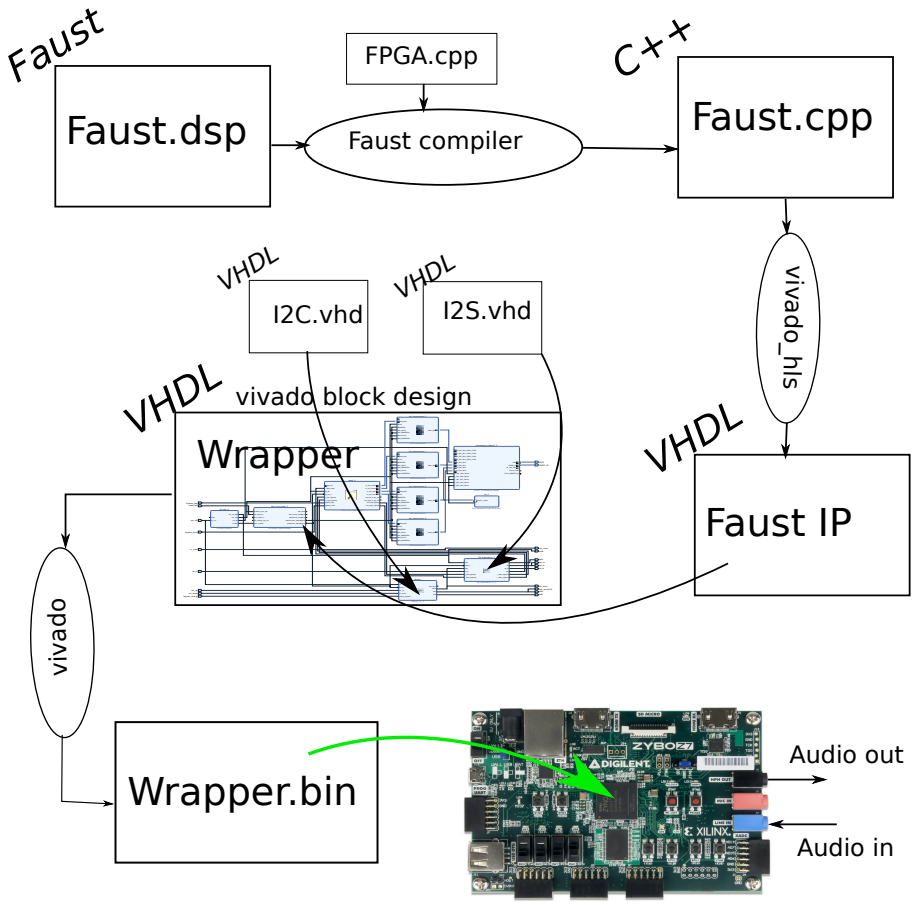

FIGURE 2.1 – Compilation flow of the first Syfala version `syfala-v1.0`o, square boxes represent files, italic skewed words reprensent their source langage, round boxes represent tools operating on files.

— There are four memory zone identified : the `i/fcontrol` are used to store control parameters, the `{i/f}zone` are used to store sample (`fzone`) or sample index (`izone`). The Faust compiler ensures coherent access in these different memory zones in the C++ generated code.
— In the first version of the compiler, for simplicity reasons, these four memory arrays are stored on the external RAM of the Zybo board and hence need and AXI bus interface IP to be accessed from the FPGA.
— the `bypass_dsp` and `bypass_faust` are used for debugging purpose (I'm not sure that they are implemented right now)

2.1 The `fpga.cpp` architecture file

The `fpga.cpp` file is the Faust *architecture file* corresponding to the FPGA target architecture (currently only Xilinx architectures are supported by syfala). It is represented

```
#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 2.2 – The *architecture* file used by Faust on syfala v1.0

on figure 2.2, it is important to understand this file.

As mentioned above, four pragmas are used to indicate that the four memory zone are all placed in external memory, accessed with a different axi bus.

The `initmydsp` is executed only once (first activation of the IP), while `controlmydsp` and `computemydsp` are executed at each samples. The underlying assumption it that the generated IP will be triggered by a `start` signal (i.e. a basic hand shake protocol), indicating that 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 bit, they are interpreted as *decimal part of signed samples between -1 and 1.*

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)	24 bits representation of c sent to i2s
0.12345678123456	0.1234567	$c = a * 2^{23} = 1035630$	[000011111100110101101110]
−0.12345678123456	−0.1234567	$c = a * 2^{23} = -1035630$	[111100000011001010010010]

2.2 Interfacing Faust, I2C and I2S

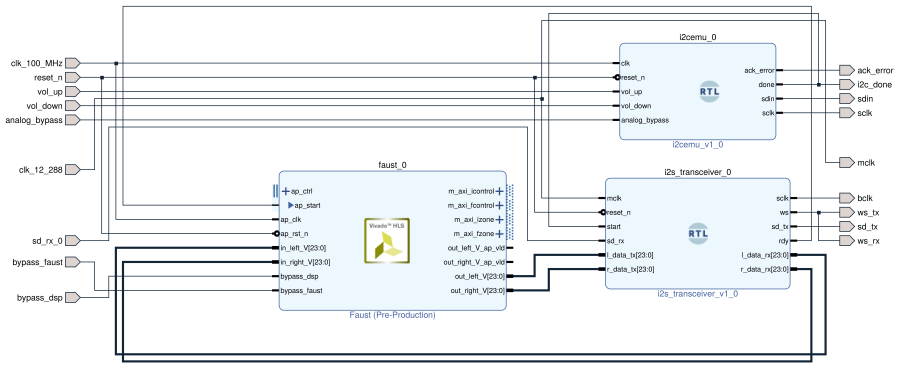

FIGURE 2.3 – The bloc design obtained by connecting Faust IP, I2C and I2S IPs

Figure 2.3 shows how the Faust IP, is interconnected with I2C and I2S IPs. All these IPs have a hardwired 125MHz system clock (i.e. 8ns clock). the `i2cemu` and `i2s_transceiver` have been written during Adeyemi Gbadamosi internship, they should be improved yet but are working for what we want to do.

2.3 The `i2cemu` IP

```
--  i2c sequences: (S=Start, A=Acknowledge (from slave), P=Stop
--  bits:     S 7 6 .... 1 0 A 15 14 ... 10 9 8 A 7 ... 1 0 A P
--  write seq:  |dev addr| 0   | reg addr   | |Reg data |
--
--  bits   S 7 6 .... 1 0 A 15 14 ... 10 9 0 A S 7 6 ...  1 0 A 7... 0 A P
--  read seq:|dev addr| 0   | reg addr   |       |dev addr| 1   |data|
```

FIGURE 2.4 – Explanation of the read and write i2c sequences as documented in `i2c_master.vhd` (see Adeyemi's report [Gba])

The `i2cemu` IP is the first one activated after reboot. It configures the registers of the `SSM2603` chip with the values of figure 2.5 (extracted from SSM2603 datasheet [Dev]). It also proposes an output volume adaptation connected to two buttons of the Zybo board (each button pressure adds/retreives a bit to the volume registers R2, R3). This IP implements

register	address	binary value	value	role
R0	0000000	000010111	23	Left channel ADC input volume (0dB, default)
R1	0000001	000010111	23	Right channel ADC input volume (0dB, default)
R2	0000010	001111001	121	Left channel DAC output volume (+0dB)
R3	0000011	001111001	121	Right channel DAC output volume (+0dB)
R4	0000100	000001100	12	Analog audio path (mic select & bypass enable ())
R5	0000101	000000000	6	Digital audio path, 48kH rate, high-pass filter
R6	0000110	000100000	32	Power management (Crystal power down), all other up (clk out, DAC, ADC, Mic, linein) (?)
R7	0000111	000001010	10	Digital audio I/F : Slave mode, 24 bits I2S mode
R8	0001000	00000000	0	Sampling rate : ADC and DAC at 48kHz
R9	0001001	00000001	1	Digital core active

FIGURE 2.5 – I2C register value initialized by the `i2c_emu` IP (see [Dev], p20), R4 and R6 to be checked.

the i2c protocol for configuring the SSM2603 register, then waits for a given period of time (75ms), and activate the start signal sent to the IPs `Faust` and `i2s_transceiver`. More details on the `i2c_emu` IP can be found in Adeyemi's master report [Gba]. The sequence followed by each i2c transaction is documented in [Dev]. In Fig. 2.4 we sketch the sequence of a read and write sequence on the i2c bus (extracted from `i2c_master.vhd` comments).

The `i2c_emu` still have to be consolidated (in particular, in current version `sda` and `scl` are not `inout` port). The `sclk` is set to 400Mhz in a generic parameter of the component.

2.4 The `i2s_transceiver` IP

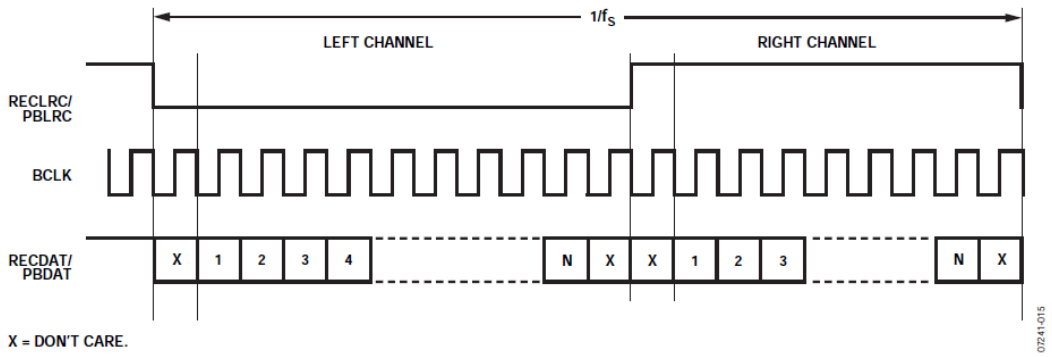

FIGURE 2.6 – I2S mode used as protocol between `i2s_transceiver` and the audio codec SSM2603 (from [Dev])

The `i2s_transceiver` is the one that really transmit the bits between the FPGA

and the audio codec. The data is serialized and transmitted/received on the `sd_tx`/`sd_rx` port to `recdat`/`pbdat` ports of the SSM2603 audio codec. The protocol used in our design (SSM26903 previously configured with register values shown in Fig. 2.5) is the one illustrated on Fig. 2.6 : the 24 bits are serially transmitted along the `bclk` clock (see also [Dev]). The `ws` signal chooses between left and right channel.

The `i2s_transceiver` is then connected to the `faust` IP as explained in Fig. 2.2. It has to be consolidated because the hand-shake protocol is not finished here, only the `ap_start` signal initiated by the `i2s_transceiver ready` signal triggers the sampling of the samples by the `faust` IP.

2.5 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 highly simplified by the use of the `Clocking Wizard` IP, which itself inputs the FPGA system clock (`sys_clk`) and output the required clocks.

FPGA system Clock : 125Mhz The *internal* FPGA clock that triggers every registers of the FPGA is depending of the complexity of the design (i.e. the complexity of the longest combinatorial path), it is called `sys_clk` on 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 [6].

Audio codec internal Master Clock : $mclk = 256 * f_s$ The clock regulating the SSM2603 should be a multiple of the sampling frequency. In the I2C configuration (Fig. 2.5), we have configured the chip to run with 48kHz sampling rate, $mclk$ to be $256 * f_s$, hence :

$$mclk = 256 * f_s = 256 * 48kHz = 12.288MHz$$

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

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 [7]

The `i2c_emu` clocks The clock used in `i2c_emu` 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 also `sclk` in Vivado bloc design, but it is not to be mislead with the `sclk` of I2S . If the FPGA system clock (125MHz) is changed, the `i2c_master.vhd` should be modified

6. to be checked yet
7. To be checked, on my design this clock is set at 100Mhz

accordingly. There is an ugly process to divide this clock once more in order to produce a `data_clk` which is shifted from `sclk` of 1/4 of a period. I do not remember exactly what was that for, but it works currently.

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 on figure 2.6) and the **ws** clock (word select) which select the left or right channel (illustrated as `reclrc/pblrc` on Fig. 2.6).

There is a fixed ratio between these two clocks and the `mclk` mentioned above :`mclk/sclk=4` (i.e. `mclk` is 4 time faster `sclk`) and `sclk/ws=64`. This is hard-coded in `i2c_transceiver` generic VHDL parameters :

```
generic(
  mclk_sclk_ratio : integer := 4;   --number of mclk periods per sclk period
  sclk_ws_ratio   : integer := 64;  --number of sclk periods per word select period
  d_width         : integer := 24);   --data width
```

Hence, one `ws` period is $T_{ws} = 4 * 64 * T_{mclk} = 256 * T_{mclk} = T_{audio} = \frac{1}{48kHz} = 20.83\mu s$.

At time of writing, when debugging the `I2S` communication, `mclk` and `sclk` are correctly running, but there is a problem with the `ws` clock : its period is not regular, sometimes it is 20 μs sometimes less, sometimes more. the `ws` signal seems to be hieratic at some points, while, given the VHDL code, it should be stable (at $20.83\mu s$).

Moreover, the left channel is never set (allways to 0), while the correct values are arriving on the right channel. We will have to check if this error comes from the HLS or from the `I2S` transmission.

3 Complete Syfala block design

The first version of of the Syfala compilation flow (provided by Adeyemi Gbadamosi and Ousmane Touat) was the one of figure 2.3. But because of the small size of the memory available in bloc RAMs on the FPGA, a simple *echo* faust function could not be compiled as there was not enough memory on the FPGA to stored the echoed samples.

Here we decided to use the external RAM present on the Zybo board, and Gero Müller set up the block design which is reprensted on Fig. 3.1 (and called `v1-four-axi`).

One can see the same interconnection of the `faust` IP with `i2c_emu` and `i2stransceiver`, but also the connections with the Zynq processing systems (which contains the AXI bus interface to the external memory) and the Four AXI controllers used for the four AXI bus accesses. The clocking wizard is on the left.

This bloc design can be manually edited once the syfala `Makefile` has been executed once (see Annex, p 11), for testing changes on the design.

This first version (`v1-four-axi`) used to many bus accesses and runs only with the `echo.dsp`. A section version (called `v2-one-axi`) keeps only one AXI for storing samples in the memory (needed for echos) and keeps the other faust input array (see Fig. 2.2) as global variable (i.e. removing I/O port for these variables). The third version (called

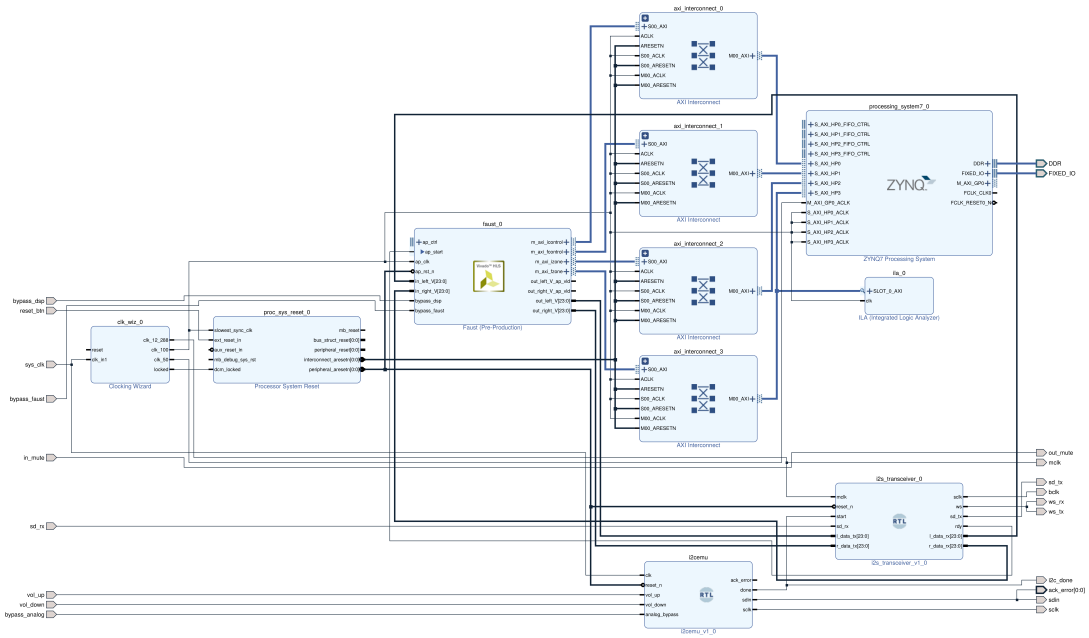

FIGURE 3.1 – The bloc design of the whole syfala project

`v3-debug-i2s`) adds i2c and i2s signal on Zybo board GPIO to be debugged with a logic analyzer.

Références

[Dev] Analog Devices. Low power audio codec ssm2603 data sheet. https ://www.analog.com/en/products/ssm2603.html. 6, 7, 8

[Gba] Adeyemi Gbadamosi. Low latency audio processing on fpga by high level synthesis. University of Burgundy, Master's Thesis. 6, 7

A Install the syfala toolchain (for syfala-v1.0)

The Syfala project (Synthetiseur Faible Latence pour FPGA) has started as a FIL project, it will probably continue for a while. The Syfala tool chain is a compilation tool chain of Faust program on FPGA (currently Xilinx Zynq present on Zybo-Z7 10 board). This document explains how to install and run the toolchain on a linux [8] which means in practice :

— Install the Faust compiler (correct version depending on the Syfala version)
— Creating a Xilinx account and downloading/installing a particular version of Xilinx `vivado` toolchain
— Clone the Syfala directory and run the basic echo example

This document explain the way to use the Syfala git repository corresponding to tag syfala-v1.0, section A.5 recalls the differences with previous Syfala version.

sections A.2, A.3 and A.4 explain how to do "by hand" what is realized when typing `make` in the `syfala` directory with Gero's scripts namely :

```
mkdir -p faust_ip
faust -lang c -light -os -a fpga.cpp -o faust_ip/faust.cpp echo.dsp
vivado_hls -f run_hls.tcl
vivado -mode batch -source build.tcl
```

A.1 Installing `vivado` and `vivado_hls`

Have a look at Yohan's slides and tutorial on http://perso.eleves.ens-rennes.fr/~yugue555/2019-vivado-hls-tutorial/.

A.1.1 Getting vivado tools (WebPack Edition)

The procedure to obtain the vivado tools :

— Open an account on https://www.xilinx.com/registration
— The Xilinx download page (https://www.xilinx.com/support/download.html) contains links for downloading the "Vivado Design Suite - HLx Edi-tions - Full Product". It is available for both Linux and Windows. When the installer prompts a choice for which version to install, select the **WebPack Edition**
— Download `Xilinx_Unified_2019.2_1106_2127_Lin64.bin`
— `chmod a+x Xilinx_Unified_2019.2_1106_2127_Lin64.bin`
— `./Xilinx_Unified_2019.2_1106_2127_Lin64.bin` (is takes one hour and **5OGB** on your hard drive)
  — choose a directory for vivado (for instance `$HOME/vivado`)
  — Update your `PATH` in `.bashrc` (for instance :
  `export PATH=$HOME/vivado/Vivado/2019.2/bin:${PATH}`)

A.1.2 Installing Vivado Board Files for Digilent Boards

Important : This step is needed to enable vivado to generate code for the Zybo Z7

8. tested on Ubuntu 18.04

Look at https://reference.digilentinc.com/learn/programmable-logic/tutorials/zybo-getting-started-with-zynq/start?redirect=1

it explains that you still have to install Vivado Board Files for Digilent Boards (Legacy). Basicaly, you gave to download a ZIP file and install it in a particular directory (for example : `$HOME/vivado/Vivado/2019.2/data/boards/board_files`)

WARNING KNOWN BUG: it is a known bug that `vivado` is sensible to the "locale" environment variable on linux, hence you have to set these variables in your `.bashrc` file :

```
export LC_ALL=en_US.UTF-8
export LC_NUMERIC=en_US.UTF-8
```

A.2 Generate `faust.cpp` with Syfala tool chain (git tag : syfala-v1.0, 7 arguments to Faust function,)

This section explains how to run the first version of syfala git repository : version which is tagged `syfala-v1.0` (version that uses the external ram to store samples). This version needs to a version of Faust greater than 19.3 (tag $b5fa8e83df111ba4f6b0ed5d6a4d5542187d3f55$ for instance) is to be used with the `2019.1` version of `vivado`

A.2.1 Clone Faust and Syfala git repositories

Syfala checkout The syfala repository is on Inria gitlab (https://gitlab.inria.fr/risset/syfala), first get an account, ask to be added to the members of the project and clone it. To clone the version needed here you can use the following commands :

```
git clone https://gitlab.inria.fr/risset/syfala.git mysyfala
cd mysyfala/v2-one-axi
make
```

or if you have installed your ssh key on gitlab :

```
git clone git@gitlab.inria.fr:risset/syfala.git mysyfala
cd mysyfala/v2-one-axi
make
```

Faust checkout Then clone the Faust repository on github (https://github.com/grame-cncm/faust), open (read mode) to everybody.

```
git clone https://github.com/grame-cncm/faust
cd faust
```

if you are not sure about the version of faust (2.20.2 for instance), you can checkout tag b5fa8e83df111ba4f6b0ed5d6a4d5542187d3f55 of Faust :

```
git checkout b5fa8e83df111ba4f6b0ed5d6a4d5542187d3f55
```

Then compile and install Faust

```
cd faust
make
sudo make install
```

Generate the `echo.cpp` file The source file is the Faust program `echo.dsp`, generating the `cpp` file that is used by `vivado_hls` is done with the `'-os'` flag of the Faust compiler (meaning 'one sample'). :

```
cd syfala
mkdir -p build/faust_ip
faust -lang c -light -os -a fpga.cpp -o build/faust_ip/faust.cpp echo.dsp
```

Launch HLS on `echo.cpp` The `vivado_hls` script is stored in file `run_hls.tcl` :

```
  cd build
  vivado_hls -f ../run_hls.tcl
```

This generates the `build/faust_ip/faust/` directory which contains the IP generated by `vivado_hls` and packaged by vivado to be recognized as an IP.

synthesize the whole design with vivado This is done by creating the vivado project using the `project.tcl script` and then running it. The compilation can be quite long (20mn) :

```
  cd build
  vivado -mode batch -source ../project.tcl -tclargs --origin_dir ".."
  vivado -mode batch -source ../build.tcl
```

A.3 From Faust to VHDL with `vivado_hls` (using .dcp format)

There are several way of exporting an IP (i.e. a circuit) that is generated by `vivado_hls`, here we use the `.dcp` format, because we could not have it working with an exported IP.

— create a directory for your synthesis : e.g. `faust_hls`
— copy the file `faust.cpp` (generated in section A.2.1) into a this new directory
— launch `vivado_hls`
→ give a project name (for instance `faust_hls`)
→ add files : `faust.cpp`
→ DO NOT add a testbench (TODO for later)
→ solution configuration : **period 8, part : xc7z010clg400-1**
→ click on finish **only once** (it takes some time here before launching `vivado_hls`)
→ select top function :
`project setting -> synthesis -> topfunction -> browse -> select 'faust'`
→ click on synthesis (i.e. play)
— check that HLS works correctly looking at the console
— Generate a `.dcp` file :
`export RTL -> Synthesized checkpoint (dcp) + VHDL`

The `faust.dcp` is exported in `$projectName/solutionName/impl/ip`. for instance : `faust_hls/faust_hls/solution1/impl/ip/faust.dcp`

A.4 From VHDL with bitstream (using .dcp format)

Again these are the instruction for using `.dcp` formated IP generated by `vivado_hls`

A.4.1 Generate the bitstream

— Create a directory for the project, for instance `faust_bistream`
— `cd faust_bitstream`
— mkdir `faust_ip`
— copy the source files needed to generate the bitstream :
  — copy the faust `.dcp` IP into the `faust_ip` directory :
    `cp -r ../faust_hls/faust_hls/solution1/impl/ip/faust.dcp faust_ip/`
  — copy the other source files (I2C and I2C VHDL files) :
    `cp -r ../src .`
— Launch vivado, select 'create a new project' (RTL)
→ Add source src/*
→ Add source `faust_ip/faust.dcp`
→ Add constraint file src/master.xdc
→ Select **board** (not part) : zybo Z7-10
→ Run synthesis
→ Run implementation
→ Generate bitstream

A.4.2 Program the board

— Plug the USB cable to Zybo
→ Open hardware manager
→ Open target (you should see xc7z01_1(1))

Install the audio matos (phone/sound source on line in, headphone on line out), and program the FPGA :

→ program device → xc7z01_1 → program

You should hear echoed music. If you do not, make sure that all switches (SW0-SW3) are in position up (towards LD0-LD3). BTN0 and BTN1 can be used to increase (resp decrease the volume)

A.5 Old stuff : Generate `faust.cpp` with first version of the Syfala tool chain (5 arguments to Faust function)

This section explains how to run the first version of the echo example (version that do not use the external ram to store samples). This version needs to get and older version of Faust, namely FAUST version 2.18.7 (26.09.2019) and correspond to a version of the Syfala Gitlab of 1/12/2019

A.5.1 Clone Faust and Syfala git repositories

Syfala checkout The syfala repository is on Inria gitlab (https://gitlab.inria.fr/risset/syfala), first get an account, ask to be added to the members of the project and clone it. To clone the version needed here you can use the following commands :

```
git clone https://gitlab.inria.fr/risset/syfala.git
cd syfala
git checkout 72f37006893d623ee69946e06b4a444da2a39548
```

Faust checkout Then clone the Faust repository on github (https://github.com/grame-cncm/faust), open (read mode) to everybody : use a version of Faust that is more recent than 2.19.3. For instance most recent version.

```
git clone https://github.com/grame-cncm/faust
cd faust
git checkout 0727ebd715da9726fe81b70d16dcb18c0382586e
```

Then compile and install Faust

```
cd faust
make
sudo make install
```

B The syfala team

Here is a list of person that have contributed to the Syfala project :

— Tanguy Risset
— Yann Orlarey
— Romain Michon
— Stephane Letz
— Florent de Dinechin
— Alain Darte
— Yohan Uguen
— Gero Müller
— Adeyemi Gbadamosi
— Ousmane Touat

C Important "ticks" to be known ! !

This section regroups all the tricks that can result in unlimited waste of time if not known

C.1 Locale setting on linux

WARNING KNOWN BUG: it is a known bug that `vivado` is sensible to the "locale" environment variable on linux, hence you have to set these variables in your `.bashrc` file :

```
export LC_ALL=en_US.UTF-8
export LC_NUMERIC=en_US.UTF-8
```

If you do not, you might end up with unpredictible behaviour of Vivado.

C.2 Installing Vivado Board Files for Digilent Boards

It is necessary, once Vivado install, to add support for new digilent board. the content of directory `board_files` has to be copied in `$vivado/2019.2/data/boards/board_files` (see https://reference.digilentinc.com/learn/programmable-logic/tutorials/zybo-getting-start start?redirect=1)

the content of `board_files` can be obtain by typing, in syfala source directory : `git submodule update --init`

C.3 Digilent driver for linux

On some linux install, programming the Zybo board will need to install an additionnal "driver" : Adept2 https://reference.digilentinc.com/reference/software/adept/start?redirect=1#software_downloads