

Getting started with the Syfala project: From Faust to FPGA

Tanguy Risset, Maxime Popoff and the Syfala Team

23 février 2022

Table des matières

0	Very Quick Start	2
1	Syfala v6.3 compilation flow	3
1.1	The Faust IP and the <code>fpga.cpp</code> architecture file	4
1.2	Interfacing Faust IP and audio codec : I2S	6
1.3	Time, Clocks and the ordering of ticks in the Syfala system	9
1.4	The ARM application software and the <code>arm.cpp</code> architecture file	10
2	A complete example : simple sinewave	12
A	Install the syfala toolchain for syfala 6.3	17
A.1	Installing Faust	18
A.2	Installing Vivado, Vitis and Vitis_hls	19
A.3	Installing Vivado Board Files and Linux drivers	19
A.3.1	Vivado Board Files for Digilent Boards	19
A.3.2	Cable drivers (Linux only)	19
A.4	Installing udev rules to use JTAG connection	20
A.5	Clone the Syfala git and compile a Syfala application	20
B	The Syfala team objectives	21
B.1	The syfala team	22
C	Known bugs : Important “tricks” to be known !!	22
C.1	Locale setting on linux	22
C.2	Save the Vivado Install file in case of installation failure	22
C.3	Vivado Installation stuck at "final processing : Generating installed device list"	22
C.4	Installing Vivado Board Files for Digilent Boards	23
C.5	Cable drivers (Linux only)	23
C.6	Digilent driver for linux	23
C.7	Vitis installation	23
C.8	"sys/cdefs.h' file not found" during vitis_HLS compilation	23

0 Very Quick Start

Last update of this document : 23 février 2022

Most recent version : Syfala v6.3 (os2) uses, Vivado 2020.2 and Faust 2.39.3

```
#make sure that vivado (v=2020.2) and Faust (v>2.39.3) are installed
#on your computer
git clone https://github.com/inria-emeraude/syfala.git my-clone-syfala
cd my-clone-syfala/syfala
make
# connect the Zybo by USB with SW0 switched on LDO side and
# blue jumper on JTAG
make boot
#listen to audio ‘‘HPH OUT’’
```

Syfala has been started almost two years ago, a first preliminary presentation was done at IFC 2020 [2]. A more recent publication should appear at SMC2022. There has been a number of *version* of Syfala, each *version* implying great changes in the sources file, and tools used hence requiring a new source code. Initial development were performed on internal Inria gitlab site (<https://gitlab.inria.fr/risset/syfala>). Since feb. 2022 a public github syfala site has been opened (<https://github.com/inria-emeraude/syfala>). The current version released (previously in directory v6.3-os2, renamed syfala in public github) makes the following choices :

- One-sample strategy : the FPGA DSP kernel is launched at each new sample and the result is available before the arrival of the next sample
- No use of pentlinux. The software running on the ARM of the Zynq SoC is used *bare-metal* : no operating system is present.
- The DDR3 memory is accessed by the FPGA DSP kernel, allowing to have long delay lines in DSP programs implemented. The DDR3 is also accessed in a *bare metal* manner : no MMU is used.
- The whole design has been optimised for low latency, efficient memory accesses, and software initialisation (see SMC2022 paper).
- The FPGA DSP kernel can be controled with a hardware interface or a software interface. The software interface is using the serial port UART between the host processor and the ARM on the Zynq. The hardware interface use SPI interface to knobs and slider, an open hardware board design is available on github/emeraude organisation).

1 Syfala v6.3 compilation flow

The required tools installations (`vivado`, `vitis`, `vitis_hls`, `Faust`) are explained in Annex A. In the cloned directory, Syfala compilation flow is deployed in a subdirectory : version v6.3 is available in directory `syfala`. Here are the ways to compile your first Faust IP :

```
git clone https://github.com/inria-emeraude/syfala mysyfala
cd mysyfala/syfala
make
```

or if you have installed your ssh key on github :

```
git git@github.com:inria-emeraude/syfala.git mysyfala
cd mysyfala/syfala
make
```

The Syfala compilation flows v6.3 follow the schematics of Figure 1.1. The default configuration when cloning syfala github code attempts to compile a Faust program called `virtualAnalog.dsp` which is present in the `mysyfala/faust` directory and is configured to use a *software* control interface (i.e. not a hardware control interface).

The parameters of the compilation launched by the `make` command can be modified in the `Makefile` file (name of DSP program compiled) or in the `configFAUST.h` file (hardware/software interface, codec used etc.). The successive commands called by the `make` command above are the following :

```
faust -lang c -light -os2 -a fpga.cpp -uim -mcd 0 -o faust_v6.cpp \
    ../faust/virtualAnalog.dsp
vitis_hls -f ../scripts/ip_v6.tcl
vivado -mode batch -source scripts/project_v6.tcl -tclargs
faust -i -lang cpp -os2 -mcd 0 -a arm.cpp ../faust/virtualAnalog.dsp \
    -o faust_v6_app.cpp
xsct ../scripts/application_v6.tcl
```

The same result can be equivalently obtained by performing each step individually with the following commands :

```
make faust /* makes the faust_v6.cpp file */
make ip /* uses vitis_hls to synthesize faust_v6.cpp */
make project /* build the faust_v6.xpr vivado project */
make bitstream /* execute the vivado faust_v6.xpr project */
make app /* create and compile the control application */
make boot /* dwnld bitstream+app on Zynq (JTAG) and boot*/
make controlUI /* launch the control UI on the host computer */
```

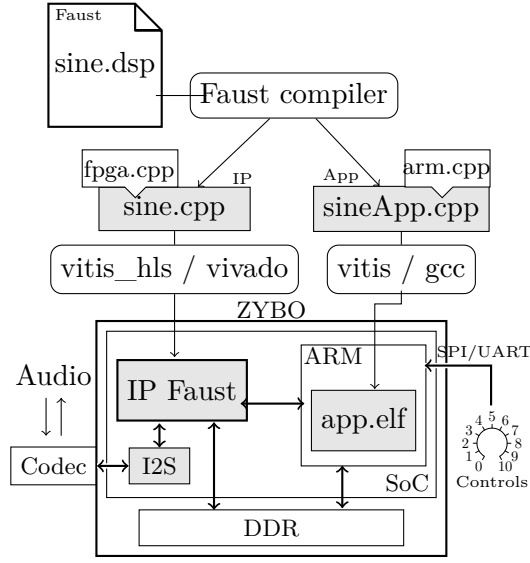


FIGURE 1.1 – Syfala compilation flow, grey boxes are generated during the compilation flow

The choices that have been made Syfala v6.3 are the following :

- Implement a *one sample* flag in the Faust compiler (`-os2`) 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` IP that will be synthesized by `vitis_hls`. This interface is present in the architecture file `fpga.cpp` detailed in Section 1.1
- Have a fixed software running on the ARM, performing constants and delays initialization and then constantly updating controllers – using hardware or software interface – and sending them to the IP. This *application* uses the `arm.cpp` architecture file and is described in Section 1.4

1.1 The Faust IP and the `fpga.cpp` architecture file

The `fpga.cpp` file is the Faust *architecture file* for Xilinx FPGA target (currently only Xilinx FPGA architectures are supported by syfala). The `fpga.cpp` determines the interface of the Faust IP. It is important to understand this interface because it highly influences many performances issues. Changing this interface is possible but it implies to change all vivado scripts present in the compilation flow, hence it requires many manual tuning before getting to new automatic compilation flow with a new interface of the Faust IP.

The interface of the Faust IP is determined by the parameter of the `faust_v6()` function which is the function synthesized by `vitis_hls`. The prototype of the `faust_v6()` function, extracted from the `fpga.cpp` file is shown in Fig. 1.2, HLS pragmas indicate

```

void faust_v6(ap_int<DATA_WIDTH> in_left_V, ap_int<DATA_WIDTH> in_right_V,
             ap_int<DATA_WIDTH> *out_left_V, ap_int<DATA_WIDTH> *out_right_V,
             FAUSTFLOAT *ram, bool *outGPIO1, bool *outGPIO2,
             bool debugSwitch, int ARM_fControl[16], int ARM_iControl[16],
             int DEBUG_toIP_tab[32], int ARM_passive_controller[32],
             int soft_reset, int ramBaseAddr, int ramDepth,
             int userVar, bool enable_RAM_access)
{
#pragma HLS INTERFACE s_axilite port=ARM_fControl
#pragma HLS INTERFACE s_axilite port=ARM_iControl
#pragma HLS INTERFACE s_axilite port=ARM_passive_controller
#pragma HLS INTERFACE s_axilite port=DEBUG_toIP_tab
#pragma HLS INTERFACE s_axilite port=soft_reset
#pragma HLS INTERFACE s_axilite port=ramBaseAddr
#pragma HLS INTERFACE s_axilite port=ramDepth
#pragma HLS INTERFACE s_axilite port=userVar
#pragma HLS INTERFACE s_axilite port=enable_RAM_access
#pragma HLS INTERFACE m_axi port=ram latency=50
    [...]
}

```

FIGURE 1.2 – Prototype of the `faust_v6()` function defined in the `fpga.cpp` architecture file. This function is synthesized by `vitis_hls` to generate the Faust IP

how each parameter of the IP are interfaced with the rest of the system. The following conventions are used (see `faust_v6.cpp` file generated in the `build/faut_ip` directory) :

- Stereo input and output (i.e. `in_left_V`, `in_right_V`, `out_left_V`, `out_right_V`) 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. The sample bit depth can be changed in the file `configFAUST.h`
- All other parameters of the IP are transmitted from the ARM processor via the s-AXI protocol, except the `ram` parameter which is the access to thr DDR3 memory.
- The DDR3 memory is accessed via the m-AXI protocol in a *bare metal* manner : a memory zone is reserved by the ARM program (explicitely reserved in the linker script) and the address and size of this zone are transmitted to the IP via the `ramBaseAddress` and `ramDepth` parameters. Note the `latency=50` pragmas which indicate that we *estimate* that a memory access will take 50 FPGA clock cycle (tuned at 120Mhz), this estimate is used by `viti_hls` to produce estimation of the timing performance of the IP (file `faust_v6.rpt`), but it is only an estimation of course.
- `ARM_icontrol[16]` and `ARM_fControl[16]` arrays are used to transmit controllers values (integer values or floating point controllers) from ARM to IP, hence in this version there must be less than 16 controllers of each type, but these can be changed before compilation of course.
- `useVar`, `DEBUG_toIP_tab[16]`, `ARM_passive_controller[16]`, `outGPIO1` and `outGPIO2`

```

void faust_v6(...)
{
if (enable_RAM_access) {
    if (cpt==0) {
        cpt++;
        /* Download initialization of constants from DDR3 content */
        instanceConstantsFromMemmydsp(&DSP,SAMPLE_RATE,I_ZONE,F_ZONE);
    }
    else {
        /* compute one sample */
        computemydsp(&DSP, inputs, outputs, icontrol, fcontrol, I_ZONE, F_ZONE);
    }
    /* Copy produced outputs, scaleFactor cast between float and ap_int */
    *out_left_V = ap_int<DATA_WIDTH>(outputs[0] * scaleFactor);
    *out_right_V = ap_int<DATA_WIDTH>(outputs[1] * scaleFactor);
}
}

```

FIGURE 1.3 – Body of the `faust_v6()` function synthesized by `vitis_hls` to generate the Faust IP

can be used for debugging purpose.

- `enable_RAM_access` is a boolean that indicates to the IP that the DDR3 initialisation performed by the ARM are finished and the the IP can start to access the DDR3.

the body of the `faust_v6()` function is shown in Fig. 1.3. The `computemydsp()` function is the function computing the effective signal processing on input/output, it is generated by the Faust compiler in the `faust_v6.cpp` file.

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 (<i>a</i>)	value truncated for 24 bits (<i>b</i>)	value stored in <code>out_left_V</code> (<i>c</i>)	24 bits representation of <i>c</i> sent to i2s
0.12345678123456	0.1234567	$c = a * 2^{23} = 1035630$	[000011111100110101101110]
-0.12345678123456	-0.1234567	$c = a * 2^{23} = -1035630$	[111100000011001010010010]

1.2 Interfacing Faust IP and audio codec : I2S

Figure 1.4 shows how the Faust IP, is interconnected with the rest of the system. All these IPs have a hardwired system clock at approximately 120Mhz (i.e. 8.33ns system clock). It is very easy and very useful to open the `vivado` project that generates the design. This can be done with the following commands :

```
make project
```

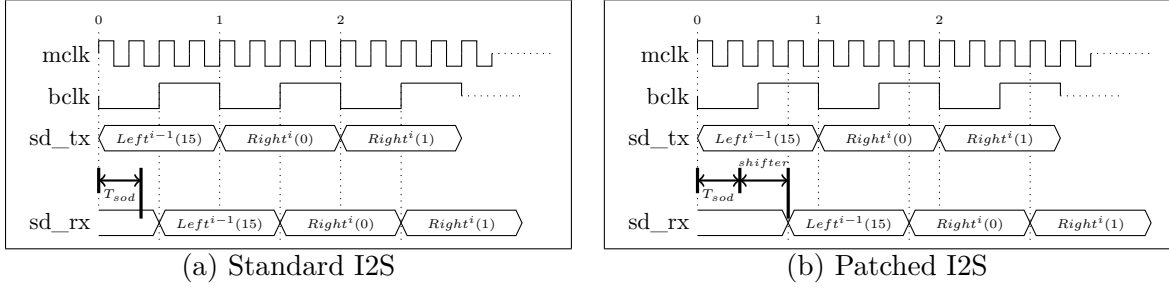



FIGURE 1.7 – The left chronogram (a) illustrates the T_{sod} time needed for the information to transit from codec to FPGA. In a standard I2S, the `sd_rx` bit is sampled on the rising edge of `bclk`. On the right (b) is illustrated our patch delaying the sampling of a `mclk` period, taking into account the time needed to transit through the level shifter

protocol from Xilinx `vitis_hls`) with the Faust IP in order to transmit and receive samples from the Faust IP. The `ap_start` signal is initiated by the `i2s_transceiver` and when the two Faust IP output are ready (`out_left_V` and `out_right_V`), the signals `out_left_V_ap_vld` and `out_right_V_ap_vld` are raised *for one system clock cycle*. A hand shake is proposed in the I2S transceiver to grab the output values when they are available (they are not necessarily available simultaneously).

1.3 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 : 120Mhz 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 **120Mhz** (i.e. setting approximately a **8.33ns** 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 : $2 \times 4 \times d_{width} \times f_s$ We call d_{width} the number of cycle needed to send the bits of one sample, remember that, as explained above : d_{width} is 16 for 16 bit-wide samples but 32 for 24 bit wide samples (and 32 for 32 bit wide samples too). The clock regulating the SSM2603 (`mclk`) should be a multiple of the sampling frequency, it should be exactly $f_{mclk} = 2 \times 4 \times d_{width} \times f_s$, where f_s is the sample rate. Indeed, as `bclk` clock will be four times slower than `mclk` clock, we will have time to send 2 samples of d_{width} bits in one sample cycle.

For instance, if we configure the chip to run with 48kHz sampling rate with 24 bit samples, f_{mclk} should be :

$$f_{mclk} = 8 \times 32 \times f_s = 256 * 48kHz = 12.288MHz$$

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**.

Vivado IP's clocks The AXI bus and the Zynq processing system require a 100Mhz clock. The Zynq processing system requires also a 50Mhz clock²

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 1.5) and the **ws** clock (word select) which select the left or right channel (illustrated as **ws** on Fig. 1.5).

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**). The ration between **sclk** and **ws** is also fixed but it depends on the bit depth of the sample : $sclk/ws = 2 \times d_{width}$. We have hard-coded these ratios in **i2c_transceiver.vhd** generic VHDL parameters which are generated at compile time, depending on the sample bit-depth choosen in the file **configFAUST.h**

For instance, at 48kHz sampling rate with 24 bit samples, one **ws** period is $T_{ws} = 4 \times 2 \times 32 \times T_{mclk} = 256 \times T_{mclk} = T_{audio} = \frac{1}{48kHz} = 20.83\mu s$. Here are the generic parameters used for this configuration in **i2s_transceiver.vhd**

```
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,

1.4 The ARM application software and the arm.cpp architecture file

Zynq FPGAs include a so-called *processing system* which consists in a complete SoC integrating *i*) a dual core ARM Cortex A9 processor, *ii*) the FPGA fabric, *iii*) high performance and general purpose buses between ARM and FPGA (s-AXI port) and *iv*) an interface to an external DDR3 memory (see Fig. 1.8). Ideally, the DSP computations should be executed on the FPGA and the control and initialization should be executed on the ARM processor. The Faust language proposes several interfaces to the user : sliders or button and even feedback information. In the remaining of this documents, we will refer to these interface devices as *controllers*.

The **faust** compiler is invoqued a second time. The first invocation has generated the **faust_v6.cpp** file used to generate the IP as in Syfala version 1 (using the **fpga.cpp**

2. To be checked, on most recent syfala version, no such clock is generated

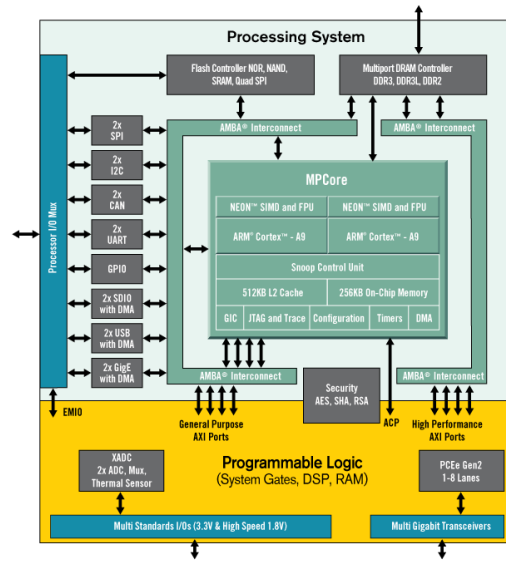


FIGURE 1.8 – Architecture of Xilinx Zynq processing system (from <https://www.rs-online.com/designspark/getting-started-with-xilinx-zynq-all-programmable-soc>)

architecture file). The second invocation is used to generate the `faust_v6_app.cpp` program that will run on the ARM (using the `arm.cpp` architecture file).

The `faust_v6_app.cpp` is quite long because it re-uses many contributions from the Faust ecosystem. Here are the actions executed by the application on the ARM processor (i.e. the actions of the `faust_v6_app.cpp` file) :

- It initializes the `ddr_ptr` pointer to the DDR memory and erases the part of the memory used by the FPGA IP. The address of the `ddr_ptr` is heritated from a macro defined in the linker script :


```
u32* ddr_ptr = (u32*)FRAME_BUFFER_BASEADDR;
```
- it initializes the `izone` and `fzone` which are then transmitted to the Faust IP :


```
izone = (int*)(ddr_ptr);
fzone = (float*)(ddr_ptr + FAUST_INT_ZONE);
```
- it initialize various peripherals of the Soc :
 - GPIOs
 - SPI peripheral (used to get controlers/sliders valuers)
 - I2C (used to configure the audio codec)
 - Faust IP
 - DDR3 memory
- It defines a user interface for the DSP program (UI)
- It defines a class `mydsp` which correspond to all the variables of the DSP program stored in the Block Rams by the Faust IP : delay lines, temporary computation, etc. This “additionnal” declaration is used to initialize some of these variables (in

- particular constants).
- It maintains a state for each controller and updates them when their values changes, either from hardware (in case of hardware interface) or from software (i.e. via the UART connection in case of software interface).
- It sends these controllers values repetitively to the Faust IP.

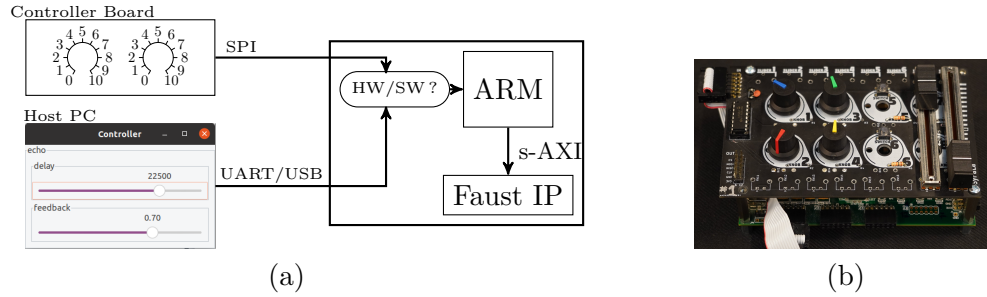


FIGURE 1.9 – (a) Interface selection between software interface (GTK app) and hardware interface (knobs such those shown in (b)). The design of the hardware board such as (b) can be freely available on github.

The `faust_v6_app.elf` file is cross-compiled to ARM binary format on the host using the cross compilation tool proposed by `vitis` from the files `faust_v6_app.cpp`, and some other files present in the `src` directory. The compilation is configured by Xilinx `xsct` tool using the script `scripts/application_v6.tcl`

Depending on the information written in the `configFaust.h` file, the code executed by `faust_v6_app.elf` launches a hardware interface to control the Faust IP or a software interface to control the Faust IP. This is shown on Fig. 1.9

2 A complete example : simple sinewave

Imagine we want to implement on FPGA a filter-based sine wave oscillator. Such a sine wave is written in Faust in Fig. 2.1. There is one controller which selects the oscillator frequency. Note the "`[knob:1]`" meta data that indicates that this controller will be associated to the first knob in case of hardware interface.

The computation of `th`, `c` and `s` are depending on the frequency value, hence we expect all these variables to be computed at control rate, hence on the ARM, not on the FPGA. On the other hand, the computation of `nlf2` is performed at each sample (sample rate) and will be implemented on the FPGA.

The first step of the compilation flow is to generate a C++ program from the Faust code, this is done by executing `make faust`. All the generated files that are related to the Faust IP are generated in the directory `build/Faust_v6`. In particular, the `faust_v6.cpp` is generated in `build/Faust_v6/Faust_v6.cpp`.

```

import("stdfaust.lib");

freq = hslider("freq [knob:1]",440,50,1000,0.01);
nlf2(f,r,x) = ((<:_,_) ,(<:_,_) :
               (*s),*(c),*(c),*(0-s)) :>
               (*x),+(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 2.1 – Filter-based sine wave oscillator in Faust used for illustrating the compilation process.

```

v6.3-ex-sin> make faust
***** Faust IP generation *****
***** ../faust/sinewave-biquad-inlined.dsp -> faust_v6.cpp *****
mkdir -p build/faust_v6_ip
faust -lang c -light -os2 -a fpga.cpp -uim -mcd 0 -o build/faust_v6_ip/faust_v6.cpp \
  ../faust/sinewave-biquad-inlined.dsp
cp configFAUST.h build/faust_v6_ip
v6.3-ex-sin>

```

An excerpt of file `faust_v6.cpp` is shown on Fig. 2.2. One can first notice the structure `mydsp` that is built for this example, the output samples are computed by the `computemydsp()` function. In this example, as the memory used is small, all variables are stored in Block Rams, hence declared here, in `faust_v6.cpp`. By looking at the body of the `faust_v6()` function (i.e. the “main” function), one can see that for the first sample, the function `instanceConstantsFromMemmydsp()` which copies the initialized constant `fconst0` on the FPGA, then the `computemydsp` is executed for all other samples. `fRec` names are usually used for delay lines, the IOTA is used to implement delay line by circular buffers.

The second step of the compilation flow is to synthesize the Faust IP from the `faust_v6.cpp` using `vitis_hls`, this is done by typing `make ip`. The IP is generated in directory `build/Faust_v6_ip/Faust_v6`. The report of the HLS, indicating the size of the resulting IP and execution time in terms of FPGA cycles can be seen by typing `make rpt`. The execution time of the HLS is approximately 1 mn :

```

[...]
```

`typedef struct {
 int fSampleRate;
 float fConst0;
 FAUSTFLOAT fHslider0;
 int IOTA0;
 int iVec0[2];
 float fRec0[2];
 float fRec1[2];
} mydsp;
[....]
void instanceConstantsFromMemmydsp(mydsp* dsp, int sample_rate, int* iZone, float* fZone) {
 dsp->fSampleRate = sample_rate;
 dsp->fConst0 = fZone[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);
}
[....]
/* body of faust_v6() function */
if (enable_RAM_access) {
 if (cpt==0) {
 /* first iteration: constant initialization */
 cpt++;
 instanceConstantsFromMemmydsp(&DSP,SAMPLE_RATE,I_ZONE,F_ZONE);
 }
 else
 {
 /* all other iterations: compute one sample */

 computemydsp(&DSP, inputs, outputs, icontrol, fcontrol, I_ZONE, F_ZONE);
 }
} else {`

FIGURE 2.2 – Excerpt of `faust_v6.cpp` C++ code generated by the Faust compiler from the Faust code presented on Fig. 2.1 when tuned for the FPGA target.

```

v6.3-ex-sin> make ip
***** HLS:  faust_v6 IP generation *****
Utilisation de Vitis_HLS
cd build && vitis_hls -f ../scripts/ip_v6.tcl

***** Vitis HLS - High-Level Synthesis from C, C++ and OpenCL v2020.2 (64-bit)
***
[...]
```

INFO: [HLS 200-802] Generated output file faust_v6_ip/faust_v6/impl/export.zip
INFO: [Common 17-206] Exiting vitis_hls at Thu Feb 3 09:24:29 2022...

```

v6.3-ex-sin>

```

The next step is to synthesize the whole design that includes the Faust IP. This is done by executing `make bitstream`. This command first builds the `Faust_v6_project.xpr`

vivado project from the TCL files and then executes it to produce the bitstream. The resulting file `main_wrapper.xsa` is generated in `build/hw_export` directory. One important point here is that the `Faust_v6_project.xpr` can be opened directly with vivado 2020.2 GUI and modified and re-synthesized. This can be usefull for exploring other block designs (other parameters for the Faust IP for instance). The `main_wrapper.xsa` is also saved in a backup directory in order to be able to use alternatively several version of the the IP without re-synthesizing. This synthesis last about 15mn for Zybo Z10.

```
v6.3-ex-sin> make bitstream
*****
***** PROJECT CREATION *****
mkdir -p build
cd build && vivado -mode batch -source ../scripts/project_v6.tcl -tclargs\
  --origin_dir $YOURDIR/v6.3-ex-sin
**** Vivado v2020.2 (64-bit)
[....]
source ../scripts/build_project_v6.tcl
[...]
***** SYNTHESIS DONE *****
**** main_wapprer.xsa generated in hw_export ****
mkdir -p backup && cp build/hw_export/main_wrapper.xsa \
  backup/sinewave-biquad-inlined_2022-02-03T10:20:44.xsa
v6.3-ex-sin>
```

Then you have to compile the application file that will run on the ARM processor. This application file is generated by the Faust compiler by using the `arm.cpp` file. Its re-uses many software components developped for the Faust ecosystem and uses also the drivers provided by Xilinx in vivado. Then the `application_v6.tcl` script is executed with `xsct` (Xilinx Software Command-line Tool) which is an an interactive and scriptable command-line interface to Xilinx `vitis` (formerly Xilinx SDK).

```

v6.3-ex-sin> make app
***** ../faust/sinewave-biquad-inlined.dsp -> faust_v6_app.cpp *
*****
faust -i -lang cpp -os2 -mcd 0 -a arm.cpp ../faust/sinewave-biquad-inlined.dsp\
-o build/faust_v6_application/generated_src/faust_v6_app.cpp
*****
***** ARM Driver code compilation:*****
xsct ./scripts/application_v6.tcl
[...]
Finished building target: faust_v6_app.elf
Invoking: ARM v7 Print Size
arm-none-eabi-size faust_v6_app.elf |tee "faust_v6_app.elf.size"
   text    data    bss     dec     hex filename
 640400    2904 2241180 2884484 2c0384 faust_v6_app.elf
Finished building: faust_v6_app.elf.size

cp build/faust_v6_application/faust_v6_app/Debug/faust_v6_app.elf build/sw_export
v6.3-ex-sin>

```

An excerpt of file `faust_v6_app.cpp` is shown on Fig. 2.3. One can see that the `mydsp` class private fields are exactly the same as the structure `mydsp` of the Faust IP (Fig. 2.2). This allow us to have coherent view of the IP, either from inside the FPGA or from the ARM processor.

One can see that the `control` method of `mydsp` on the ARM processor (Fig. 2.3) corresponds to the computations of variables `th`, `c` and `s` of the Faust program of Fig. 2.1. As we expected, the control rate computations are executed on the ARM. Then the structure `ARMcontroller` defines the functions `sendControlToFPGA()` and `controlFPGA()`.

The function `sendControlToFPGA()` is using Xilinx driver functions for accessing m-AXI port of the Faust IP (here `fControl` and `iControl` ports). The function `controlFPGA()` will first call `fDSP->control()` in order to get new values of the controllers from the hardware or software user interface, then it will call `sendControlFPGA()` to send these values to the Faust IP.

Finally the “main” program of `Faust_v6_app.cpp` can be seen on Fig. 2.4, it is an infinite loop calling permanently `ARMcontroller->controlFPGA()`

TODO: explain make controlUI

Références

- [1] Analog Devices. Low power audio codec ssm2603 data sheet. <https://www.analog.com/en/products/ssm2603.html>. 8
- [2] 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. 2


```

[...]
```

```

class mydsp : public one_sample_dsp_real<float> {

private:

    int fSampleRate;
    float fConst0;
    FAUSTFLOAT fHslider0;
    int IOTA0;
    int iVec0[2];
    float fRec0[2];
    float fRec1[2];

public:
[...]
```

```

    virtual void control(int* RESTRICT iControl, float* RESTRICT fControl, int* RESTRICT iZone, float* RESTRICT fZone) {
        fControl[0] = fConst0 * float(fHslider0);
        fControl[1] = std::sin(fControl[0]);
        fControl[2] = std::cos(fControl[0]);
    }
[...]
```

```

}

struct ARMController {
    // Control
    ARMControlUIBase* fControlUI;
    // DSP
    mydsp* fDSP;
    [...]
void sendControlToFPGA()
{
    XFaust_v6_Write_ARM_fControl_Words(&faust_v6, 0,(u32*)fControl, FAUST_REAL_CONTROLS);
    XFaust_v6_Write_ARM_iControl_Words(&faust_v6, 0,(u32*)iControl, FAUST_INT_CONTROLS);
}

void controlFPGA()
{
    // Compute iControl and fControl from controllers value
    fDSP->control(iControl, fControl, iZone, fZone);
    // send iControl and fControl to FPGA
    sendControlToFPGA();
}
[...]
```

FIGURE 2.3 – Excerpt of C++ code generated by the Faust compiler from the Faust code presented on Fig. 2.1 when tuned for the ARM application target.

A Install the syfala toolchain for syfala 6.3

The Syfala toolchain is a compilation toolchain of Faust program on FPGA (currently Xilinx Zynq xc7z010clg400-1 present on Zybo-Z10 board). This document explains how to install and run the toolchain from version 6.3 and up (i.e. version without petalinux), on a linux³ machine. In practice, installing the Syfala tool chain means :

- Installing the Faust compiler, see section A.1 below.
- Creating a Xilinx account and downloading/installing a version 2020.2 of Xilinx vivado toolchain : vitis_hls, vivado and vitis. See section A.2 below.
- Installing Vivado Board Files for Digilent Boards, see section A.3
- Installing udev rules to use JTAG connection, see section A.4
- Cloning the Syfala directory and running a simple example as explained in Section 2.

Warning : You need approximately 50GB of disk space to install the tool chain, and a good connection. The installation take several hours. If the installer prompts a choice for which version to install, select the **Webpack Edition**

Warning all the tools of Vivado come with shell scripts that set up your \$PATH to use

3. tested on Ubuntu 18.04 and Ubuntu 20.04

```

[...]
```

```

// main program infinite loop infinite loop
void run()
{
    while (true) {
        //check if reset btn is pressed
        if (XGpio_DiscreteRead(&gpio, 1))
        {
            // IP and Zynq reset
            [....]
        }
        else
        {
            controlFPGA(); //send controllers value to IP
            fControlUI->update(); //get new controller values
        }
    }
}

```

FIGURE 2.4 – Excerpt of C++ code generated by the Faust compiler from the Faust code presented on Fig. 2.1 when tuned for the ARM application target.

them. It is quite dangerous to source them in the `.bashrc` file because it provides older version of important utilities (such as `cmake` for instance). We strongly advise you to use a function defined in your `.bashrc` file such as the following :

```

function use_vitis
{
    source $myXilinxToolDirectory/Vivado/2020.2/settings64.sh
    source $myXilinxToolDirectory/Vitis_HLS/2020.2/settings64.sh
    source $myXilinxToolDirectory/Vitis/2020.2/settings64.sh
}

```

A.1 Installing Faust

It is recommended to clone Faust from the github repository : <https://github.com/grame-cncm/faust> :

```

git clone https://github.com/grame-cncm/faust faust
cd faust
make
sudo make install

```

If you are using older version of Syfala, you might need to use older version of Faust (see `version` files in Syfala directory). the procedure is to get the commit number of the version you need here : <https://github.com/grame-cncm/faust/releases>. For instance, if you use Syfala v5.4, it requires Faust version 2.31.1 (at least), its commit number is : 32a2e92c955c4e057d424ab69a84801740d37920, then execute :

```

cd faust
git checkout 32a2e92c955c4e057d424ab69a84801740d37920
make
sudo make install

```

A.2 Installing Vivado, Vitis and Vitis_hls

If you encounter a bug during the installation, please see Section C.

The procedure is the following

- 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 Editions - Full Product". It is available for both Linux and Windows. Download the Linux installer⁴
- Download `Xilinx_Unified_2020.2_1118_1232_Lin64.bin`
- `chmod a+x Xilinx_Unified_2020.2_1118_1232_Lin64.bin`
- `./ Xilinx_Unified_2020.2_1118_1232_Lin64.bin` (it takes one hour and **100GB** on your hard drive). We suggest to use the "Download Image (Install Separately)" option. It creates a directory with a `xsetup` file to execute for installing that you can reuse in case of failure during the installation. See Section C.2.
- Launch `xsetup`
- Choose to install Vitis (it will install `vivado`, `vitis` and `vitis_hls`), It will need 110GB of disk space. If you uncheck Ultrascale, Ultrascale+, Versal ACAP and Alveo acceleration platform, it uses less space and still work.
- Agree with everything and choose a directory to install
- Install and wait for hours
- Setup the `use_vitis` function as explained above.
- Install missing Vivado board files for Digilent boards and drivers for linux (explained in Section A.3 below).

A.3 Installing Vivado Board Files and Linux drivers

A.3.1 Vivado Board Files for Digilent Boards

Important : This step is needed to enable vivado to generate code for the Zybo Z10

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). Basically, you have to download a ZIP file and install it in a particular directory (for example : `$HOME/vivado/Vivado/2020.2/data/boards/board_files`)

A.3.2 Cable drivers (Linux only)

For the Board to be recognized by the Linux system, it is necessary to install additional drivers. See <https://digilent.com/reference/programmable-logic/guides/install-cable-drivers>

4. https://www.xilinx.com/member/forms/download/xef.html?filename=Xilinx_Unified_2020.2_1118_1232_Lin64.bin

A.4 Installing udev rules to use JTAG connection

The udev rules must be configured to use the JTAG connection through USB. The udev rules configuration files are provided by Vivado, they must be installed manually on your Linux.

Here are the action to perform on Linux :

- Get the 52-digilent-usb.rules file in directory :
Vivado/2020.2/data/xicom/cable_drivers/lin64/
- Copy it (as sudo) in /etc/udev/rules.d

That's it, you should be able to program the Zybo (using `make standalone_boot` in syfala or using the hardware manager in Vivado IDE).

A.5 Clone the Syfala git and compile a Syfala application

The syfala repository is freely accessible (reading only) on github (<https://github.com/inria-emeraude/syfala>), you have to have a github account of course to clone it. As mentioned before, there may be several sub-directories with different version of Syfala (i.e. different interface for Faust hardware IP). Here are the steps needed to run Syfala :

1. **Clone Syfala github :** to clone the version needed and compile a first architecture you can use the following commands :

```
git clone https://github.com/inria-emeraude/syfala.git mysyfala
```

2. **Configure the parameter of your compilation action.** A “Syfala application” in V6.3 is from a unique source .dsp Faust program that should be chosen in the Makefile. Other parameters can be tuned in the `configFAUST.h` file.

- **choose the DSP to compile.** The Makefile can perform all the steps automatically, the .dsp and .c file are indicated at the beginning. For instance, in the Makefile in the subdirectory syfala :

```
mysdsp?=virtualAnalog
DSP = ../faust/$(mysdsp).dsp
```

- **Choose the parameter of the compilation flow:** The application that will run on the Zybo ARM – used to control the sliders of the .dsp file – is generated automatically. The parameters that can be changed are located in the `configFAUST.h` file they consist of :
 - Type of control : hardware or software (default)
 - use DDR (default) or not
 - sample rate (default 48000Hz)
 - sample bit depth (default 24 bits)
 - output volume (for headphone by default)

- **Generate hardware IP main_wrapper.xsa.** Go in the syfala directory and type `make` :

```
cd syfala
make
```

- **This will take approximately 15 minutes**

the `main_wrapper.xsa` file containing the Faust hardware IP is in directory `build/hw_export`. The files to be compiled on the ARM are in directory `build/sw_export`

3. **Program the Zybo.** Download the configuration on the Zybo : connect the Zybo by USB, make sure that the jumper are correctly positionned (see `README.md` file)

`make boot`

And listen to the line output.

B The Syfala team objectives

The Syfala project (*Synthetiseur Faible Latence pour FPGA*) has started as a FIL project, it will probably continue for a while. This document explains the technical choices that have been made on the first versions 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 installation of the toolchain itself is explained in Annex here (from p 17).

The objective is to compile Faust⁵ 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 current syfala version is able to reach a latency of $191\mu s$ with the intergrated Analog Device SSM2603 codec and a latency $11.1\mu s$ with a more efficient codec (Analog Device ADAU 1787).

Few examples of professional FPGA-based real-time audio DSP systems (i.e., Antelope Audio,⁶ Korora Audio,⁷ 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), `CatapultC` (from Mentor Graphics), but other tools are proposed today to bridge the gap between algorithmic representation and hardware level representation of a computation⁸. This project has been launched by the Emeraude team⁹ which is a collaboration between Grame research department¹⁰ and Citi laboratory.

5. <https://faust.grame.fr/>

6. <https://en.antelopeaudio.com>

7. <https://www.kororaudio.com>

8. See https://en.wikipedia.org/wiki/High-level_synthesis for instance

9. <https://team.inria.fr/emeraude/admin>

10. <https://www.grame.fr/recherche>

B.1 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
- Luc Forget
- Antonin Dudermel
- Maxime Popoff
- Thomas Delmas
- Oussama Bouksim

C Known bugs : Important “tricks” 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 unpredictable behaviour of Vivado.

C.2 Save the Vivado Install file in case of installation failure

Vivado installation tends to fail. To avoid having to redownload the installation file each time you try , we suggest to use the “ Download Image (Install Separately)” option. It creates a directory with a xsetup file to execute for installing. But don’t forget to duplicate the installation file, because Vivado will delete the xsetup installation file you use if you choose to let him delete all files after the installation failed.

C.3 Vivado Installation stuck at "final processing : Generating installed device list"

If the install of Vivado is stuck at "final processing : Generating installed device list", cancel it and install the libncurses5 lib :

```
sudo apt install libncurses5
```

C.4 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-started-with-zynq/start?redirect=1#
```

The content of `board_files` can be obtain by typing, in syfala source directory :

```
git submodule update --init11
```

Or directly here : <https://github.com/Digilent/vivado-boards>

C.5 Cable drivers (Linux only)

For the Board to be recognized by the Linux system, it is necessary to install additional drivers. See <https://digilent.com/reference/programmable-logic/guides/install-cable-drivers>

C.6 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

C.7 Vitis installation

Warning Apparently the installation process does not end correctly if the `libtinfo-dev` package is not correctly installed (<https://forums.xilinx.com/t5/Installation-and-Licensing/Installation-of-Vivado-2020-2-on-Ubuntu-20-04/td-p/1185285>). In case of doubt, execute these commands (april 2020) :

```
sudo apt update
sudo apt install libtinfo-dev
sudo ln -s /lib/x86_64-linux-gnu/libtinfo.so.6 /lib/x86_64-linux-gnu/libtinfo.so.5
```

C.8 "sys/cdefs.h' file not found" during vitis_HLS compilation

If Vitis HLS synthesis fails with the following error :

```
'sys/cdefs.h' file not found: /usr/include/features.h
```

You have to install the `g++-multilib` lib

```
sudo apt-get install g++-multilib
```

11. TODO : check, ça marche ça ?)