# Developper documentation for the Syfala project: From Faust to FPGA

The Syfala Team

26 janvier 2023

## Table des matières

# 0 Very Quick Start

Last update of this document : 26 janvier 2023

**Most recent version :** Syfala v7, Vivado 2020.2 and Faust $\geq$ 2.39.3

```
#make sure that vivado (v=2020.2) and Faust (v>2.39.3) are installed
#on your computer (see Syfala install documentation)
git clone https://github.com/inria-emeraude/syfala.git  my-clone-syfala
cd my-clone-syfala/
./syfala.tcl install
# connect the Zybo by USB with all switchs down (i.e. opposite to
# ethernet connector plug) on LD0 side and blue jumper on JTAG
syfala examples/virtualAnalog.dsp --reset
#This will compile the ‘‘example/virtualAnalog.dsp’’ (~15mn)
# --reset option is useful if you need to recompile it
syfala flash
#listen to audio ‘‘HPH OUT’’
syfala GUI
#Now you can control the virtualAnalog Synthesizer
```

Syfala has been started in 2020 [**?**, **?**]. There has been a number of *version* of Syfala, each *version* implying great changes in the source files, 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 is v7, named simply `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 petalinux. The software running on the ARM of the Zynq SoC is used *bare-metal* : no operating system is present.
- The external DDR memory is accessed by the FPGA DSP kernel, allowing to have long delay lines in DSP programs implemented. The DDR is also accessed in a *bare metal* manner : no MMU is used.
- The whole design has been optimized for low latency, efficient memory accesses, and software initialization (see [**?**]).
- The FPGA DSP kernel can be controlled with a hardware interface or a software interface. The software interface is using the UART serial port between the host processor and the ARM on the Zynq. The hardware interface uses SPI interface for knobs and sliders. An open hardware board design is available on github/emeraude organisation).

# 1 Syfala v7 compilation flow

The installation of the required tools (`vivado, vitis, vitis_hls, Faust`) is explained in the Syfala install documentation [1].

The Syfala v7 compilation flows follows the schematics of Figure **??**. When cloning syfala github, Faust programs are located in the `examples` directory, the compilation flow is configured by default to use a *software* control interface (i.e. not a hardware control interface) and to use the onboard audio codec (SSM2603 on Zybo Z7, ADAU1761 on Genesys).

Since version 7 of Syfala, the `syfala.tcl` script is used to launch the different Syfala commands. The command `./syfala.tcl install` will install in `/sur/local/bin` (as root) a `syfala` command that basically run the `./syfala.tcl` script. If you are to clone another instance of Syfala, make sure to run the `'syfala.tcl install'` command again before using it.

All Syfala generated files are produced in the `build` directory The sub-directories of the `syfala` repository are the following

```
.
|-- README.md
|-- build     // contains all the files generated by Syfala
|-- doc       // Syfala documentation
|-- examples  // Faust .dsp file
|-- include   // include files for Syfala
|-- misc      // misc (e.g. patches)
|-- scripts   // All tcl scripts
|-- source    // All sourcse files used by Syfala
|-- testbenches // VHDL testbenches (outdated now)
|-- tests     // used for testing syfala (for dev. only)
`-- tools     // higher level tools using Syfala (for dev. only)
```

The compile-time parameters added with the `syfala` command will select both the way the audio DSP will be compiled (e.g sample rate, sample bit width) and the hardware interface (e.g. codec used). The successive commands called by the command : `syfala examples/virtualAnalog.dsp` command are the following :

```
TODO: update once commands are highlighted in the script
  faust -lang c light -os2 -a fpga.cpp -uim -mcd 0 -o syfala.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 syfala_application.cpp
  xsct ./scripts/application_v6.tcl
```

1. https://github.com/inria-emeraude/syfala/blob/main/doc/dependencies.md

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

```
syfala clean / removes the build directory /
syfala examples/virtualAnalog.dsp --arch /* uses faust to generate
   HW (syfala_ip.cpp) and (syfala_application.cpp) files */
syfala --ip  /* uses vitis_hls to synthesize syfala_ip.cpp */
syfala --project /* build the syfala_project.xpr vivado project */
syfala --syn     /* execute the vivado syfala_project.xpr project
    and build the bitstream */
syfala --app /* create and compile the control application on PC */
syfala --flash /* download bitstream+app on Zynq (JTAG) and boot*/
syfala --gui /* launch the control UI on the host computer */
syfala --report /*  prints HLS report  */
```

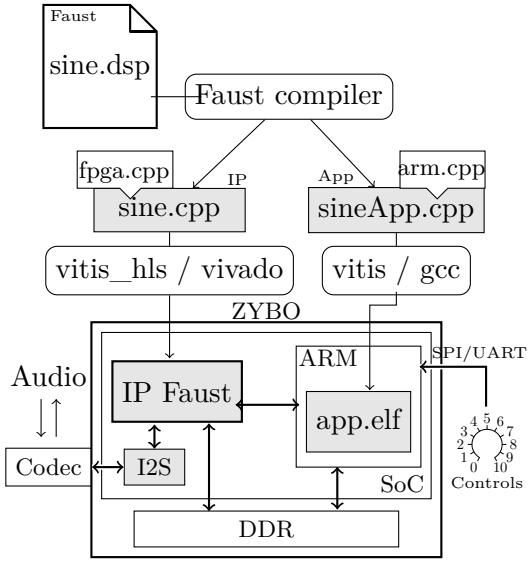



FIGURE 1.1 – Syfala compilation flow, grey boxes are generated during the compilation flow

The choices that have been made Syfala v7 are the following :

— Implement a *one sample* flag in the Faust compiler (`-os2`) that generates a `computemydsp()` function (in the CPP file generated by Faust) 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`. Despite this fixed interface, any number of controllers (i.e. sliders) can be used in the Faust program. This interface is present in the architecture file `fpga.cpp`

```
void syfala(
        sy_ap_int in_ch0_V,
        sy_ap_int in_ch1_V,
        sy_ap_int* out_ch0_V,
        sy_ap_int* out_ch1_V,
        bool *outGPIO, bool debugBtn, bool mute, bool bypass,
        int ARM_fControl[9],
        int ARM_iControl[2],
        int ARM_passive_controller[2],
        FAUSTFLOAT *ram, int ramBaseAddr, int ramDepth, 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=ramBaseAddr
#pragma HLS INTERFACE s_axilite port=ramDepth
#pragma HLS INTERFACE s_axilite port=enable_RAM_access
#pragma HLS INTERFACE m_axi port=ram latency=50
[...]
```

FIGURE 1.2 – Prototype of the `syfala()` function defined in the `fpga.cpp` architecture file for a stereo Input/Output DSP program. This file is generated from a template to adapt to the actual number of codecs used in our system architecture. This function is synthesized by `vitis_hls` to generate the Syfala IP

detailed in Section **??**

— 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 **??**

## 1.1 The Syfala 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 Syfala IP. It is important to understand this interface because it highly influences many performance 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 Syfala IP.

The interface of the Syfala IP is determined by the parameters of the `syfala()` function which is the function synthesized by `vitis_HLS`. The prototype of the `syfala()` function, extracted from the `Syfala_ip.cpp` file is shown in Fig. **??**, HLS pragmas indicate how each parameter of the IP is interfaced with the rest of the system. The following conventions are used (see `Syfala_ip.cpp` file generated in the `build/syfala_ip` directory) :

— Stereo input and output (i.e. `in_ch0_V`, `in_ch1_V`, `out_ch0_V`, `out_ch1_V`) are 24

bit wide signed integer interpreted as a value between -1 and 1, which are to be sent and received from the I2S transceiver which himself will interface with the audio codec. The sample bit depth and the number of Input/Output channels can be changed via syfala parameters.

— All other parameters of the IP are transmitted from the ARM processor via the `axilite` protocol[2], except the `ram` parameter which is the access to the DDR memory.

— The DDR memory is accessed via the 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 approx. 120Mhz), this estimate is used by `vitis_hls` to produce estimation of the timing performance of the IP (file `syfala.rpt` in directory `./build/syfala_ip/syfala/syn/report/`), but it is only an estimation of course.

— `ARM_icontrol[9]` and `ARM_fControl[2]` arrays are used to transmit controllers values (integer values or floating point controllers) from ARM to IP. Again the '9' and '2' values are generated from the DSP audio file.

— `ARM_passive_controller[2]`, `outGPIO`, `mute`, `bypass` can be used for debugging purpose.

— `enable_RAM_access` is a boolean that indicates to the IP that the DDR initialization performed by the ARM is finished and that the IP can start to access the DDR.

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

The `scaleFactor` value (i.e. `8388607.0f`) is exactly $2^{23} - 1 = (1 \ll (23)) - 1$. If 24 bitwidth sample are used, The input/output of the `syfala` 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 bitwidth are configure in the syfala command which generates the file `build/include/syconfig.hpp`.

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_ch0_V` ($c$) | 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] |

## 1.2 Interfacing Faust IP and audio codec : I2S

Figure **??** shows how the Faust IP, is interconnected with the rest of the system. All these IPs have a hardwired system clock at 122.88Mhz (i.e. approx. 8 ns system clock). It

2. Throughout the document, we will refer to `axilite` for the `axilite 4` protocol used for IP parameter (sometimes called s-axilite) and `AXI` for `axi 4` protocol used to access the DDR memory (sometimes called m-axi)

```
void syfala([...])
{
if (enable_RAM_access) {
    if (cpt==0) {
      cpt++:
      /* Download initialization of constants from DDR content */
      instanceConstantsFromMemmydsp(&DSP,SYFALA_SAMPLE_RATE,I_ZONE,F_ZONE);
    }
    else {
        /* compute one sample */
        computemydsp(&DSP, inputs, outputs, icontrol, fcontrol, I_ZONE, F_ZONE);
        sendToARM(ARM_passive_controller);
      }
  /* Saturate  outputs, scaleFactor cast between float and ap_int */
     for(int i=0; i<FAUST_OUTPUTS; i++){
     if (outputs[i]> 1.0) outputs[i]=1.0;
     else if (outputs[i]< -1.0) outputs[i]=-1.0;
    }
    *out_ch0_V = sy_ap_int(outputs[0] * scaleFactor);
    *out_ch1_V = sy_ap_int(outputs[1] * scaleFactor);
  }
}
```

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

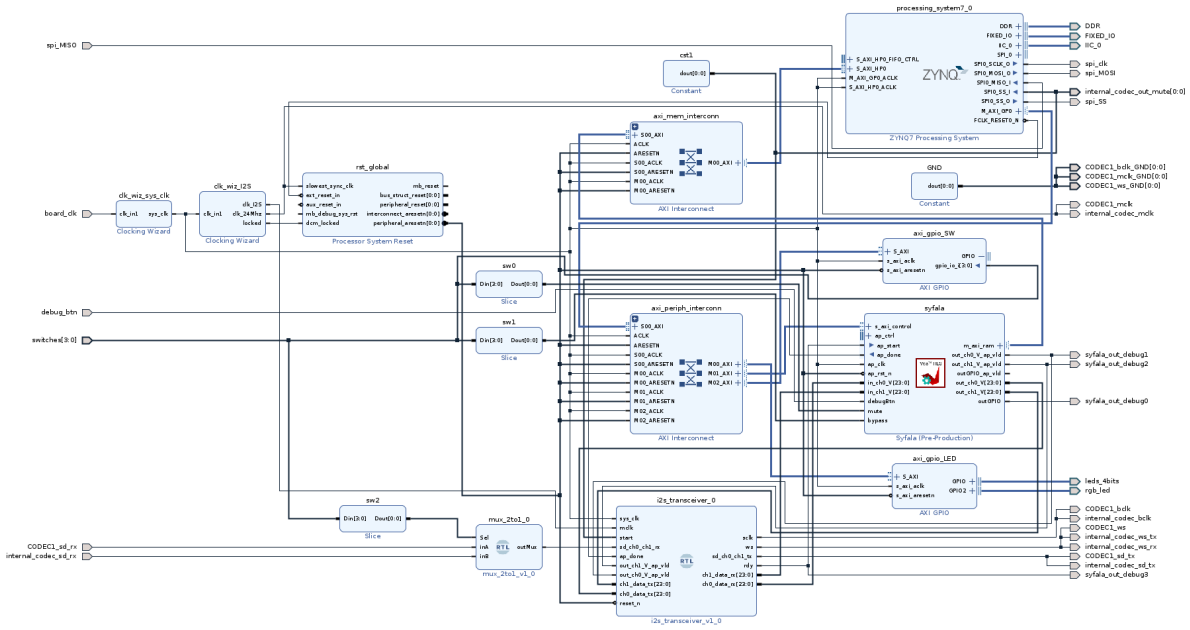



FIGURE 1.4 – The bloc design obtained by connecting Syfala IP,(Syfala v7), with I2S IPs and AXI interface to DDR

is very easy and very useful to open the `vivado` project that generates the design. This can be done with the following command (after the build is done) :

```
syfala open-project
```

Then the block design shown on Fig. **??** can be opened using `open Block Design`. One can see that the audio input/output streams of the Syfala IP are directly connected to the I2S IP (`i2s_transceiver` block), one can also see the `AXI` IP interface which is used to access DDR and the `axilite` IP interface used for interface with ARM processor. The I2S IP is in turn directly connected to I/O of the Zynq with the following convention :

- The first two channels (Ch0 and Ch1) are connected to the pad of the onboard codec (SSM2603 for ZYBO, ADAU1761 for Genesys).
- The first two channels (Ch0 and Ch1) are duplicated on GPIO pads for the use of an external codec.
- All additional channels (if existing) are connected to the GPIO pads for the use of an external codec.

The onboard codec is configured from the ARM processor as described in section **??**

The `i2s_transceiver` is the one that really transmits the bits between the FPGA and the audio codec. The data is serialized and transmitted/received on the `sd_tx`/`sd_rx` port to the ports of the audio codec. The protocol used in our design is the one illustrated on Fig. **??**, it can be configured to send 16, 24 or 32 bit-wide sample. For 16 bit configuration the sample cycle time is exactly divided in 32 cycles to transmit the $2 \times 16$ bits (left and

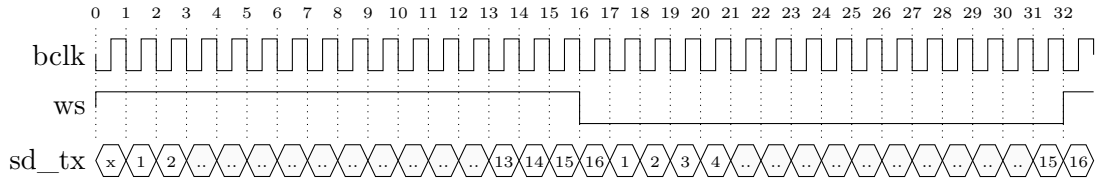



FIGURE 1.5 – I2S protocol implemented `i2s_transceiver.vhd`, between the Syfala IP and the audio codec with 16-bit samples. The `ws` signal select from left or right channel. The `sd_tx` bit stream corresponds to the 16 bits of the sample. it is shifted of 1 clock cycle from `ws` changes. `bclk` stands for *bit clock* and `ws` stands for *word select*.

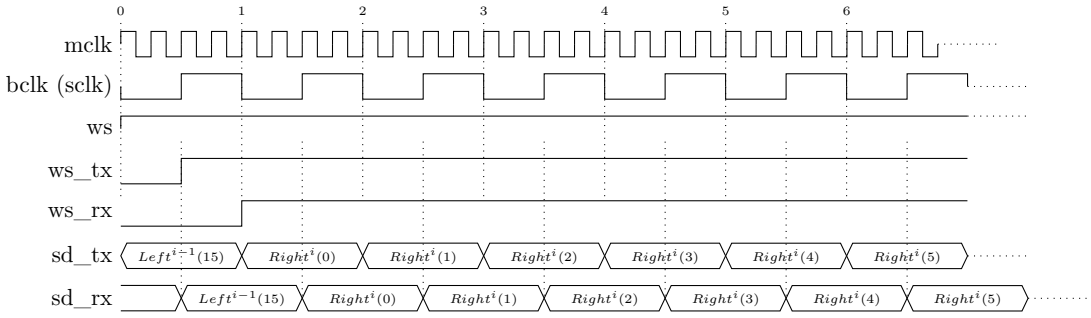



FIGURE 1.6 – Zoom on the beginning of a right sample (sample number $i$) first bits transmission : `mclk` is 4 time faster than `bclk`. `ws_tx` and `ws_rx` are delayed version of `ws`, used to synchronize starting of samples bits transmission. `sd_tx` is *produced* by the I2S IP as an output on the falling edge of `bclk` and `sd_rx` is *read* as an input on the rising edge of `bclk`.

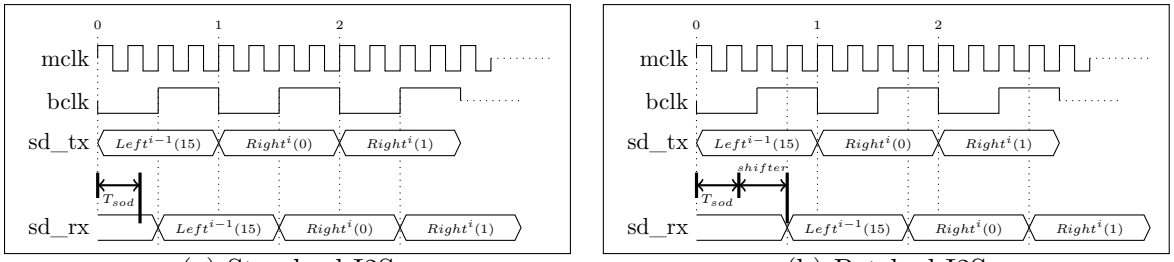
right samples), as shown on Fig. **??**. But for 24 bit-wide sample, the sample cycle is not divided in 48 (=2 × 24), but in 64 cycles as it is for 32 bit-wide samples. The sample bits are serially transmitted along the `bclk` clock as shown in Fig. **??** (see also [**?**]). The `ws` signal indicates whether current bits belong to left or right channel. However, as indicated in Fig. **??**, there is a shift of 1 cycle : the first bit send after `ws` clock fall-down is not the first bit of current left sample, it is the last bit of the previous right sample. [3]

**Syfala I2S patch** In a normal transmission, the `sd_tx` bit is positioned on the falling edge of `bclk` clock, it is transmitted from our (master) I2S to the (slave) I2S of the codec. Simultaneously, the slave I2S is positioning the `sd_rx` bit – which is *his* `sd_tx` – to be transmitted from the codec to our I2S. The `sd_rx` bit is effectively read by our I2S on the rising edge of `bclk`, this allows time for the signal to arrive through the connection between the codec and the FPGA, this time is called `Tsod` in analog device ADAUs codecs for instance (see Fig. **??**-(a) for illustration).

In our design, we have used external codecs that allows internal clock as fast as 768kHz.

3. See for instance https://www.sparkfun.com/datasheets/BreakoutBoards/I2SBUS.pdf

We have noticed that, as we needed a level shifter to adapt power supplies between the codec and the Zybo, this half a bclk cycle time may be less than the time needed for `sd_rx` to stabilize. Hence we proposed a *patch* that delays of one `mclk` cycle in addition to the half `bclk` cycle shown on Fig. **??**-(b).




(a) Standard I2S (b) Patched I2S

FIGURE 1.7 – The left chronogram (a) illustrates the `Tsod` 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

We have implemented the I2S protocol in VHDL (file `src/i2s_transceiver.vhd`). It can be parameterized by the sample bit depth as well as by the sample rate.

The `i2s_transceiver` is connected to the `Syfala` IP. It performs a hand shake (`ap_hs` protocol from Xilinx `vitis_hls`) with the Syfala IP in order to transmit and receive samples from the Syfala IP. The `ap_start` signal is initiated by the `i2s_transceiver` and when the two Syfala IP outputs are ready (`out_ch0_V` and `out_ch1_V`), the signals `out_ch0_V_ap_vld` and `out_ch1_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 two `Clocking Wizard` IP. The first clocking wizard inputs the external clock (`sys_clk`) and outputs the FPGA system clock `board_clk` and the second one outputs `mclk` and another clock at 24MHz needed by the codecs. The reason for using *two* clocking wizard instead of one is that exact frequencies for the three clocks cannot be obtained with only one clocking wizard, we need two MMCM/PLL.

**FPGA system Clock : `sys_clk` at 122.88MHz** 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 follow two rules to set this clock :

— `sys_clk` can be as fast as wanted as long as it met timing constraints.

— `sys_clk` and `mclk` should be multiples to facilitate the timing closure and minimize the negative slack.

`mclk` is a multiple of 48kHz and $f_{mclk}$=12.288MHz at 48kHz sampling rate (see below). So we usually impose `sys_clk` clock to be **122.88MHz** (i.e. setting a **8.13ns** clock when creating `vivado` and `vivado_hls` projects). Faster clocks have been tested with Syfala and should work too.

**I2S Transceiver Master Clock : `clk_I2S` at $2 \times 4 \times d_{width} \times f_s$** 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 transceiver (`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 want an I2S signal at 48kHz sampling rate with 24 bit samples, $f_{mclk}$ should be :

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

**Codec system clock : `clk_24Mhz` at 24.576MHz** The last generated clock is the system clock needed by the codecs to works. It's configurable on each codec and has no effect on the sampling rate. We use a **24.576MHz** clock which is compatible with all our tested codecs and ensure the best performances.

*In practice, the clocking wizard is not able to obtain exactly the these frequencies (because of the limitation of a PLL) so the real sample frequency obtained is not exactly 48kHz. But we ensure that all frequencies are multiples specifying the nearest synthesizable frequency. For exemple, on ZYBO we have :* $f_{sys\_clk}$=122.885835MHz, $f_{clk\_I2S}$=12.2885835MHz so $f_s$=48.002279kHz

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

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 ratio 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 `i2s_transceiver.vhd` generic VHDL parameters which are generated at compile time, depending on the sample bit-depth given as options to the `syfala` command (24 by default).

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
```

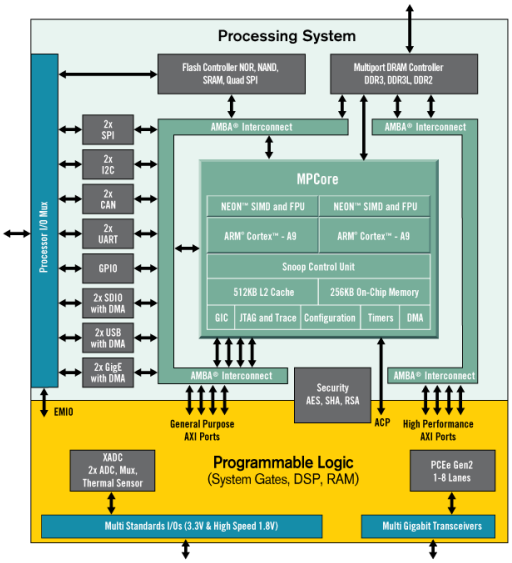



FIGURE 1.8 – Architecture of Xilinx Zynq 7000 (ZYBO) processing system (from https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html)

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

Zynq SoCs include a so-called *processing system* which consists in a dual-core ARM Cortex-A9 for Zynq7000 SoC for ZYBO, or a Quad-core ARM Cortex-A53 on Ultrascale+ MPSoC for Genesys ZU-3EG. These SoC also embed a high performance and general purpose buses between ARM and FPGA (axilite port) and an interface to an external DDR memory (see Fig. **??** and Fig. **??**).

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 invoked a second time. The first invocation has generated the `syfala.cpp` file used to generate the IP (using the `fpga.cpp` architecture file). The second invocation is used to generate the `syfala_application.cpp` program that will run on the ARM (using the `arm.cpp` architecture file).

The `syfala_application.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 `syfala_application.cpp` file) :

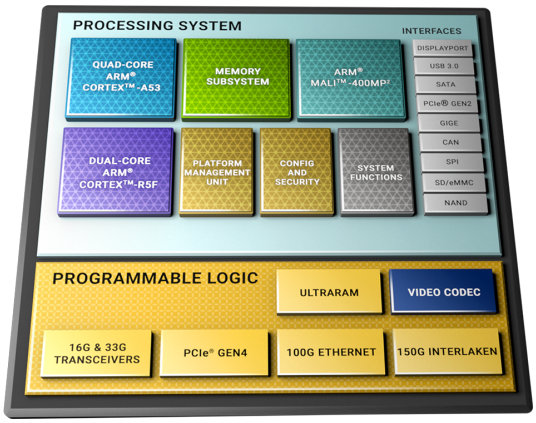



FIGURE 1.9 – Architecture of Xilinx UltraScale+ (Genesys) processing system (from https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html)

- 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 inherited 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 initializes various peripherals of the Soc :
  - GPIOs
  - SPI peripheral (used to get controlers/sliders valuers)
  - I2C (used to configure the audio codec)
  - Faust IP
  - DDR 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 "additional" 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.

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

Depending on the information of the syfala command, the code executed by `syfala_application.elf` launches a hardware interface to control the Faust IP or a software interface to control the

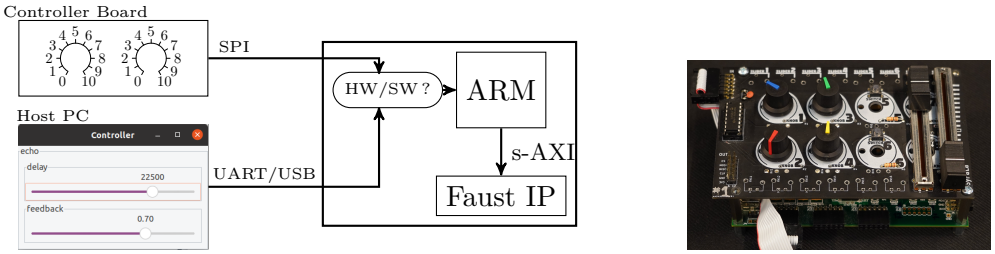



FIGURE 1.10 – (a) Interface selection between software interface (GTK app) and hardware interface (knobs such those shown in (b)). The design of the hardware board such as (b) can be freely available on github.

Faust IP. This is shown on Fig. **??**.

## 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. **??**, it is available in Syfala repository as program `sinewave-biquad-inlined.dsp` of the `examples` directory. 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.

```
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 2.1 – Filter-based sine wave oscillator in Faust used for illustrating the compilation process (file `sinewave-biquad-inlined.dsp` in `examples` directory).

The whole compilation can be done using the command :
`syfala examples/sinewave-biquad-inlined.dsp`
but we will detail the different steps.

The first step of the compilation flow is to generate a C++ program from the Faust

code, this is done by executing :
`syfala examples/sinewave-biquad-inlined.dsp --arch --reset`
this command will generate `syfala_ip.cpp` and `syfala_application.cpp` files. All the generated files are generated in the directory `build/`. The `--reset` options is mandatory if another project has already been compiled in the `build` directory. Warning, the `--reset` option will erase your previous compiled Syfala project.

The `--arch` option generates the `syfala_ip.cpp` file in directory `build/syfala_ip/` and the `syfala_application.cpp` file in `build/syfala_application/`. Note that a file `build/include/syconfig.hpp` is also created in which the parameters of the current design flow are saved (sample rate, board used, hard or software controller, etc.). From now on the `build/sinewave-biquad-inlined.dsp` will be the default DSP program syfala is working on, hence the name of the `.dsp` file do not have to be recalled at each syfala command.

```
syfala-github$ syfala examples/sinewave-biquad-inlined.dsp --arch --reset
[ INFO ] Running syfala toolchain script (v7) on Linux (5.4.0-126-generic)
[...]
[ INFO ] Generating Faust IP from Faust compiler & architecture file
[  OK  ] Generated /home/trisset/technical/syfala-github2/build/\
      syfala_ip/syfala_ip.cpp
[ ... ]
[  OK  ] Generated /home/trisset/technical/syfala-github2/build/\
      syfala_application/syfala_application.cpp
[ ... ]
[ INFO ] Script has been running for 00 minutes and 00 seconds
[  OK  ] Successful run!
```

An excerpt of file `syfala_ip.cpp` is shown on Fig. **??**. 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 `syfala_ip.cpp`. By looking at the body of the `syfala() function` (i.e. the "main" syfala IP function), one can see that, at the very beginning, the function `instanceConstantsFromMemmydsp()` is executed (it 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 `syfala_ip.cpp` using `vitis_hls`, this is done by typing `syfala --ip`. The IP is generated in directory `build/syfala_ip/syfala`. 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 `syfala report`. 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 syfala() 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);

      }
  }
[...]
```

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

```
syfala-github> syfala --ip
[ INFO ] Running syfala toolchain script (v7) on Linux (5.4.0-126-generic)
[....]
[ INFO ] Running Vitis HLS on file [...]/syfala-github/scripts/hls.tcl
****** Vitis HLS - High-Level Synthesis from C, C++ and OpenCL v2020.2 (64-bit)
[...]

[ INFO ] Script has been running for 00 minutes and 43 seconds
[  OK  ] Successful run!
syfala-github>
```

The next step is to synthesize the whole design that includes the Faust IP. For that, we need to build the bloc design. Usually, it's done a first time with the Vivado GUI and can be exported in .tcl or .vhd "Bloc Design" file to script the synthesis of the project.

But we choose to write a script called `syfala_maker.tcl` that will directly generate this "Bloc Design" file. The advantage is that the bloc design can be dynamically changed. For example, it allow us to change the number of I2S channels on the transceiver and adapt the number of used GPIO and the internal routing just with a macro. We couldn't do such a thing with a fixed block design file.

Then, the synthesize of the project is done by executing `syfala --project` and then `syfala --syn`. The first command builds the `syfala_project.xpr` vivado project from the TCL files. The second command loads and then executes the `syfala_project.xpr` project in vivado to produce the bitstream. As a result, a file `main_wrapper.xsa` is generated in `build/hw_export` directory, it corresponds to an archive containing both the FPGA bitstream and configuration of the *processing system* (i.e. the ARM subsystem). One important point here is that the `syfala_project.xpr` can be opened directly with Vivado 2020.2 GUI (by executing `syfala --open-project`) and modified and re-synthesized. This can be useful for exploring other block designs (other parameters for the Faust IP for instance). The `syfala --export` command allows you to save all generated files in an `export` directory in order not to loose them when executing a command with `--reset` option.

```
syfala-github$ syfala --project
[ INFO ] Running syfala toolchain script (v7) on Linux (5.4.0-126-generic)
[...]
[ INFO ] Running Vivado on file /home/trisset/technical/syfala-github2/build/
     sources/project.tcl
****** Vivado v2020.2 (64-bit)
[...]
[ INFO ] Script has been running for 00 minutes and 26 seconds
[  OK  ] Successful run!

syfala-github$ syfala --syn
[ INFO ] Running syfala toolchain script (v7) on Linux (5.4.0-126-generic)
[...][ INFO ] Running Vivado on file /home/trisset/technical/syfala-github2/
   scripts/synthesis.tcl
****** Vivado v2020.2 (64-bit)
[...]
Waiting for synth_1 to finish...
[...]
[ INFO ] Script has been running for 08 minutes and 58 seconds
[  OK  ] Successful run!
syfala-github>
```

Then you have to compile the application file that will run on the ARM processor. This application file has been generated by the Faust compiler at the first step (i.e. `--arch` option). It uses the `arm.cpp` architecture file as main file. Its re-uses many software components developed for the Faust ecosystem and uses also the drivers provided by Xilinx in `vivado`. Then the `application.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).

```
syfala-github> syfala --app
[ INFO ] Running syfala toolchain script (v7) on Linux (5.4.0-126-generic)
[...]
[ INFO ] Compiling Host control application
make -C ps7_cortexa9_0/libsrc/xilffs_v4_4/src -s include "SHELL=/bin/sh" "
    COMPILER=arm-none-eabi-gcc"
   "ASSEMBLER=arm-none-eabi-as" "ARCHIVER=arm-none-eabi-ar" "COMPILER_FLAGS= -O2
       -c"
   "EXTRA_COMPILER_FLAGS=-mcpu=cortex-a9 -mfpu=vfpv3 -mfloat-abi=hard -
       nostartfiles -g -Wall -Wextra"
[...]
5:11:05 Build Finished (took 1s.713ms)
Finished building projects
[  OK  ] Finished building host application
[  OK  ] Copied application sources and .elf output to sw_export directory
[ INFO ] Script has been running for 00 minutes and 43 seconds
[  OK  ] Successful run!
[  OK  ] To see the build's full log: open 'syfala_log.txt' in the repository's
  root directory
syfala-github>
```

An excerpt of file `syfala_application.cpp` is shown on Fig. **??**. One can see that the `mydsp` class private fields are exactly the same as the structure `mydsp` of the Faust IP (Fig. **??**). 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. **??**) corresponds to the computations of variables `th`, `c` and `s` of the Faust program of Fig. **??**. 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 `s-axilite` 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. `sendControlFPGA()` uses the API provided by Xilinx to communicate between the ARM and the FPGA IP (`XSyfala_Write_ARM[...]()` functions)

Finally, the generated program can be transferred to the FPGA board with the `syfala --flash` command (the Zybo board has to be plugged on a USB port of course, and a headphone should be used to hear the sounds). The control GUI is compiled with the `syfala --gui` command, and then executed with `syfala gui`. Once flashed, the `syfala_application` is launched automatically on the ARM and the bitstream is executing. The ARM has to boot first, so the `enable_RAM_access` port (see figure **??**) is used to indicate that the Syfala IP can start its computations.

```
[...]
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()
  {
    XSyfala_Write_ARM_fControl_Words(&xsyfala, 0,(u32*)fControl, FAUST_REAL_CONTROLS);
    XSyfala_Write_ARM_iControl_Words(&xsyfala, 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. **??** when tuned for the ARM application target.

# A Known bugs : Important "tricks" to be known ! !

This section regroups all the tricks that can result in unlimited waste of time if not known. These *known bugs* have been kept as they have been initially written, even if some of them do not occur anymore in more recent tool version.

## A.1 Locale setting on linux

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.`

## A.2 Patch 2022 date bug

`Vivado and Vitis tools that use HLS in the background are also affected by this issue. HLS tools set the ip_version in the format YYMMDDHHMM and this value`

```
[...]
// main program infinite loop infinite loop
  void run()
  {
    while (true) {
      //check if reset btn is pressed
    while (true) {
         controlFPGA();
         UIhandler();
         fControlUI->update();
    }
  }
[...]
```

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

is accessed as a signed integer (32-bit) that causes an overflow and generates the errors below (or something similar).

Follow this link: https://support.xilinx.com/s/article/76960?language=en_US

Download the file at the bottom of the page and unzip it in your Xilinx base install directory (Xilinx file where you have your Vitis,Vitis_HLS and Vivado files).

DONT FOLLOW THE README... Just check the "Known Issues:" section on the Xilinx page which takes over the readme.

From the Xilinx directory, run:

- export LD_LIBRARY_PATH=$PWD/Vivado/2020.2/tps/lnx64/python-3.8.3/lib/
- Vivado/2020.2/tps/lnx64/python-3.8.3/bin/python3 y2k22_patch/patch.py

## A.3 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.

## A.4 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
```

## A.5 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_

(see

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

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

### A.6 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

### A.7 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

### A.8 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
```

### A.9 "'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

### A.10 Board files: version 1.0 or 1.1?

Digilent updated his board file repository (mentioned above in section ??) and unfortunately changes the version of the board from 1.0 to 1.1. This change must be reverted because it is not taken into account in past version of vivado.

It you have a message like:

```
source /home/romain/reps/syfala/build/sources/project.tcl -notrace
ERROR: [Board 49-71] The board_part definition was not found for
  digilentinc.com:zybo-z7-10:part0:1.0. The project's board_part property was
 not set, but the project's part property was set to xc7z010clg400-1.
 Valid board_part values can be retrieved with the 'get_board_parts'
 Tcl command. Check if board.repoPaths parameter is set and the board_part
 is installed from the tcl app store.

   You should do the following:
    - go into directory:
      Vivado/2020.2/data/boards/board_files/zybo-z7-10/A.0
    - Edit the file 'board.xml' and change
      <file_version>1.1</file_version>
      into
      <file_version>1.0</file_version>
    - (Same thing for Z20 if you use Z20).
```

# 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
- Luc Forget
- Antonin Dudermel
- Maxime Popoff
- Thomas Delmas
- Oussama Bouksim
- Pierre Cochard