Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

System Construction Course 2019,

Assignment 11

Felix Friedrich, ETH Zürich, 2019

Multicore Computing on FPGAs – The Active Cells Model.

Lessons to Learn

- Understand the features and limits of a message passing architecture.
- Understand how software code can be mapped to and deployed on a multicore architecture on FPGA.
- Understand how configurable hardware can be adapted to satisfy software requirements.

Preparation

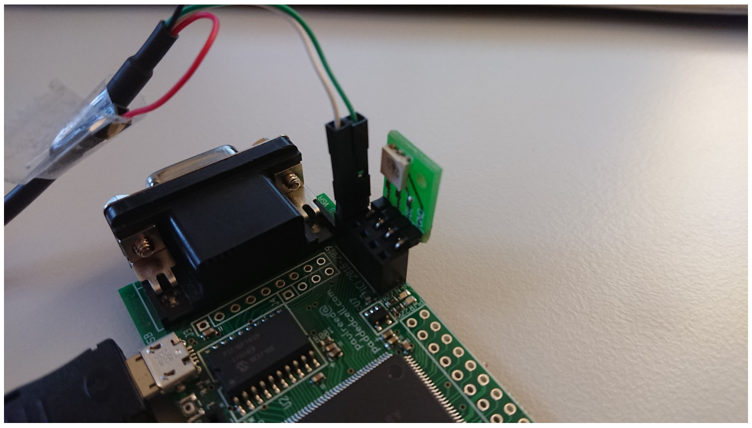
1. Update your repository or checkout the exercise from assignments/assignment11
2. Open a console in `assignment11`
3. Extract the development system:

```
unzip bin.zip source.zip
```

4. Compile the HDL tools and all hardware components with

```
./oberon execute BuildTools
```

5. Connect the board with LED inserted via USB cable to your PC.

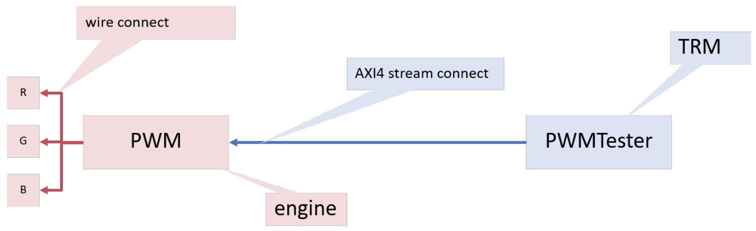

LED connection (ignore the white and green serial cables).

and try out the very first application example module Application/PwmTest.Mod, a module testing the Pulse Width Modulator hardware using

```
./oberon execute BuildPWM
```

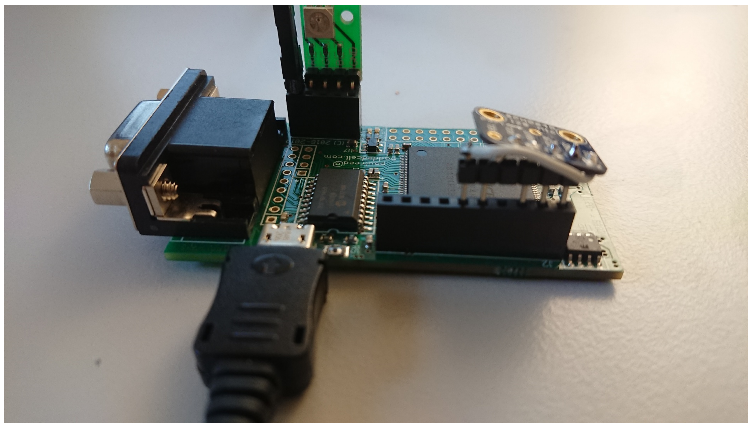
After synthesis, place and routing and deployment, the three-color LED should slowly go from red over green to blue.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich



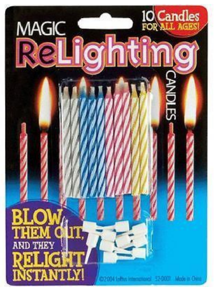
PWM Tester Network.

6. Connect the I2S board as shown on the picture below.



I2S microphone connection.

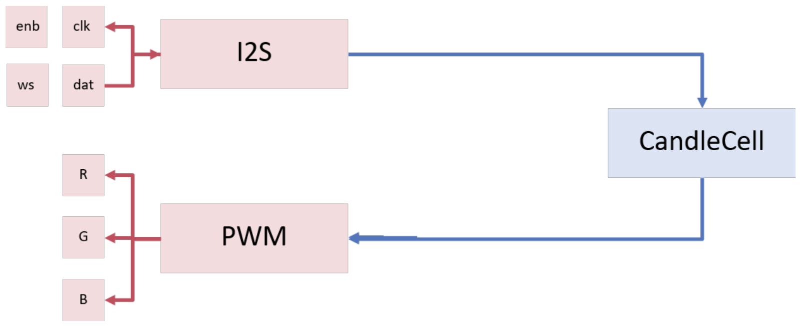
1 Candle



As an introductory exercise, we use the microphone (connected via I2S to the FPGA board) in order to detect when someone blows on the board and magically let the LEDs turn off and turn them on again after a while.
Module Application/Candle.Mod contains a network that connects the output of the I2S module to a software component which has access to a Pulse Width Modulator (PWM) driving an RGB LED.
Complement the implementation of the `CandleCell` in Candle.Mod such that it simulates a relighting candle.

Run the build script for the candle using the command

```
./oberon execute BuildCandle
```

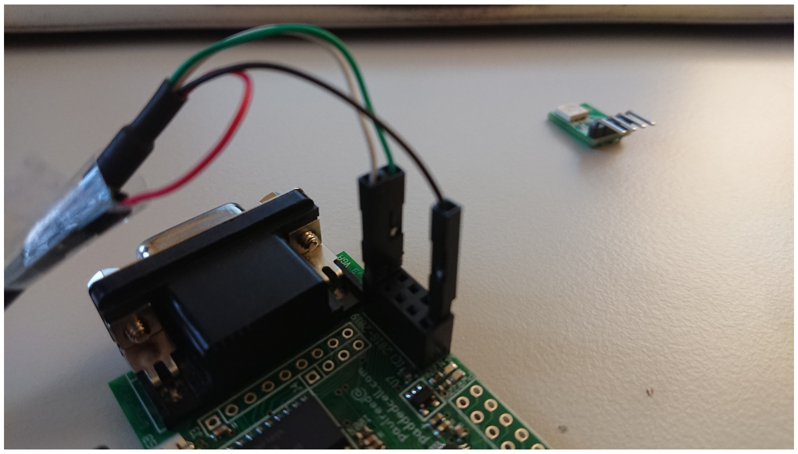


Relighting Candle Network.

2 Sampling Audio

In the previous example, you implemented the logic of a cell in a network of cells. In this example, the business logic of the cell is already given in module Application/SoundSampler.Mod and you need to construct a useful network of cells.

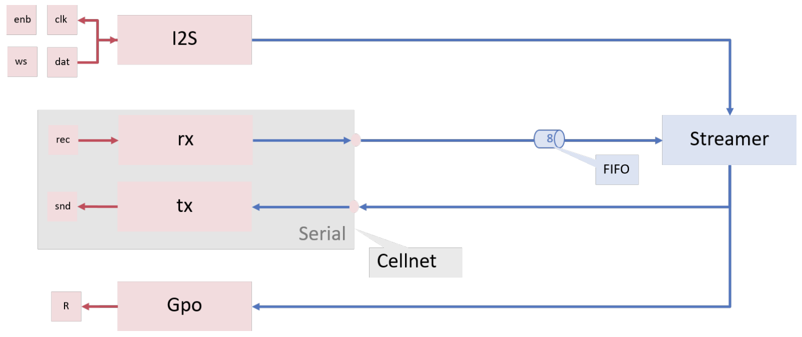
Connect the Serial-To-USB cable with the board as shown on the figure below.


I2S serial connection.

Implement the network (as displayed below) in the `Network` cellnet in module `SoundSampler.Mod`.

Run the build script for the audio sampler

```
./oberon execute BuildSampler
```



Audio Sampler Network.

The sampler sends each sample taken using a serial connection to the host PC , waiting for each character to be sent. In order to synchronize the communication (very crudely), 8192 4-byte data points will only be sent from the device to the host PC when a '0' character has been sent via serial connection from the host to the device. Sending is indicated by a light up blue LED on the board.

You can receive data on the host PC by calling

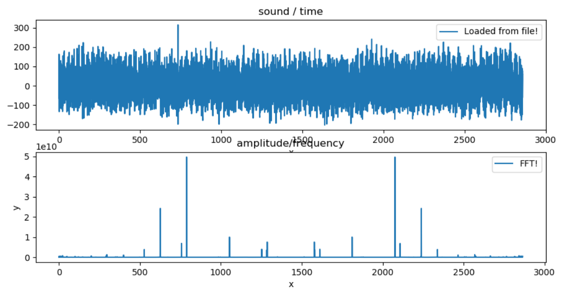
```
V24.Map 9 /dev/ttyUSB0 ~
SoundStream.Record 9 audio.txt ~
```

within the Oberon shell, or equivalently call

```
./oberon run RecordSoundStream
```

The command `Record` in Oberon module Application/SoundStream.Mod also does an FFT of the data and tries to identify the dominant audio frequencies. The data can be visualized using the python script plot_audio.py using the command

```
python3 plot_audio.py
```



Originally, it was planned to use the FFT on the hardware device in this exercise but for reasons of time and simplicity, this idea has been dismissed. You are very much welcome to implement this on the FPGA hardware, nevertheless.

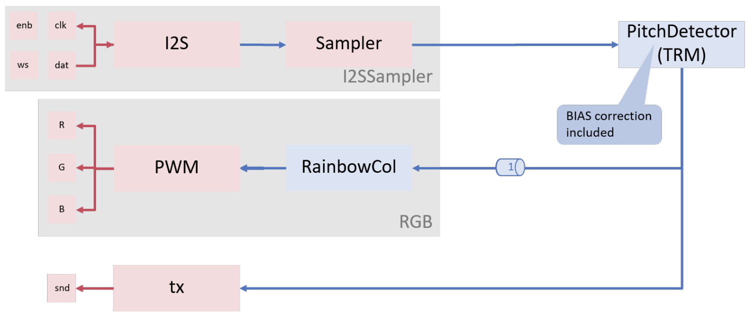
While doing the experiments for this exercise, we have found out that already counting the numbers of crossings from below the mean to above the mean in an audio signal can be used in order to estimate the base frequency observed by the device, which gave us the idea for the following part of this exercise.

3 Tuner

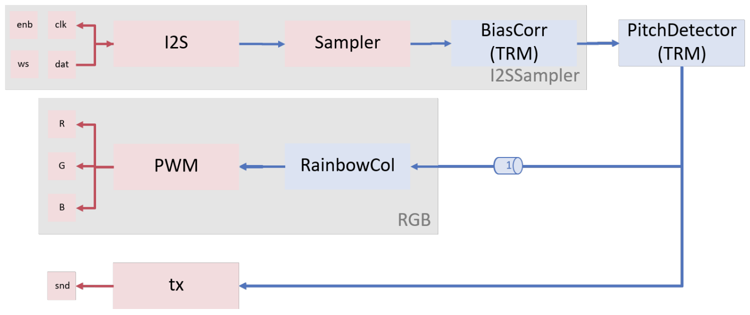
The objective of this part of the exercise is to build a system that computes the base frequency of a tone (e.g. of a musical instrument) in realtime and outputs the currently measured frequency (via serial connection) and indidates the difference to a target frequency on the RGB LED (red: too low, blue: too high, green: ok).
In order to be able to finish this in reasonable time, we provide you with our current solution where one component is missing: the bias correction.
You can alternatively use the serial connection in order to output sampling values on your host PC or the LED. Leave the green and white cables of the serial cable connected and connect the black cable (ground) alternatingly with the LED. We have only one ground pin.

Add the BIAS correction in three steps:

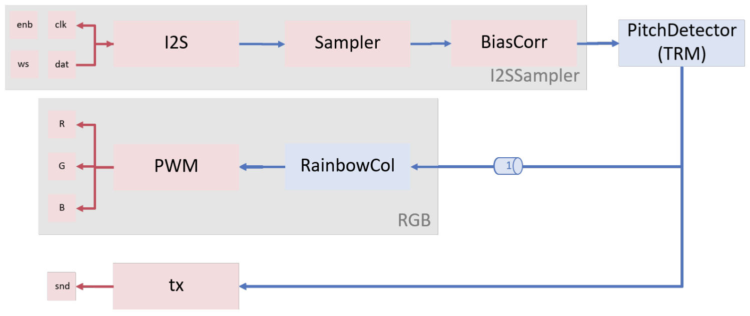
1. Compute the (integer) average value of a reasonable number of samples in software within the PitchDetector and subtract this value from each new sampled value. The (hardware) component `Sampler` provides (at most) 20000 samples per second: an attempt to read a value from the `Sampler` component output blocks until $50\mu s$ have passed since the most recent read. The sampler delivers 24 bit signed integer values (with a prevision of 18 bits, cf. I2S component data sheet).



2. Build a software BiasCorrector component that takes data from the Sampler component, applies the bias correction, and passes the corrected values on to the Pitch detector (as seen below in the network of the tuner).



3. Translate the software BiasCorrector to Verilog Hardware Description Language and provide a new specialized ActiveCells BiasCorrector engine that does this in hardware.



Steps 1 and 2 can be carried out directly in module Application/Tuner.Mod and do not require any changes in the Active Cells Hardware library. Recompile, build and deploy the module using the command

```
./oberon execute BuildTuner
```

The third step is more involved and usually consists of the following tasks

(a) Create an Active Cells registration module of the hardware module. This has already been done and can be found in module HardwareLibrary/IO/BiasCorrector.Mod. Please have a look in this module that contains the description of the software (Axi4Stream) ports and the hardware ports involved.

(b) Create the corresponding Verilog module. We have prepared a template of this module (such that it adheres to the right kind of interface) with HardwareLibrary/IO/BiasCorrector.v. The business logic of this module is largely missing.

(c) Define the hardware ports that are used by (instances) of this module for the underlying hardware / board. This definition (here in HardwareLibrary/OPALBoard.Mod) is not necessary here because the module does not provide any hardware ports.

(d) Register the Verilog module in the Active Cells Hardware library. We have carried this out already in file HardwareLibrary/Specifications.txt

The implementation of the Verilog module is non-trivial because of the Axi4Stream ports involved. It is best to have a look at other hardware modules in the Hardware library. In order to make it doable in the amount of time given, our code can already pass on unaltered data. The code can be understood with the following rules in mind.

Sender perspective:

(i) Availability of data is signaled on the output port (signal `out_tvalid`) only when data are available.

(ii) Data keeps being available on the output as long as they have not been picked up (signal `out_tready`).

Receiver perspective:

(iii) Data is only read from the input when it is available (signal `in_tvalid`).

(iv) Sender of the data sees that you have picked up the data (signal `out_tready`).

Documents

- Slides from the lecture homepage.
- I2S microphone SPH0645LM4H datasheet documents/microphone/SPH0645LM4H-B.pdf
- AMBA 4 AXI4-Stream Protocol in documents/ActiveCells/IHI0051A_amba4_axi4_stream_v1_0_protocol_spec.pdf
- Active Cells Paper in documents/ActiveCells/ComputeModelHPSoCFPGA.pdf