Pyha

Release 0.0.1

Gaspar Karm

Apr 03, 2017

Contents:

Chapter 1

Introduction

Essentially this is a Python to VHDL converter, with a specific focus on implementing DSP systems.

Main features:

- Simulate in Python. Integration to run RTL and GATE simulations.
- Structured, all-sequential and object oriented designs
- Fixed point type support(maps to VHDL fixed point library)
- Decent quality VHDL output (get what you write, keeps hierarchy)
- Integration to Intel Quartus (run GATE level simulations)
- Tools to simplify verification

Long term goal is to implement more DSP blocks, especially by using GNURadio blocks as models. In future it may be possible to turn GNURadio flow-graphs into FPGA designs, assuming we have matching FPGA blocks available.

Working principle

As shown on Fig. 1.1, Python sources are turned into synthesizable VHDL code. In `__init__`, any valid Python code can be used, all the variables are collected as registers. Objects of other classes (derived from HW) can be used as registers, even lists of objects is possible.

In addition, there are tools to help verification by automating RTL and GATE simulations.

Listing 1.1: this.py

```
print('Explicit is better than implicit.')
```

See on Listing 1.1 shows print statement

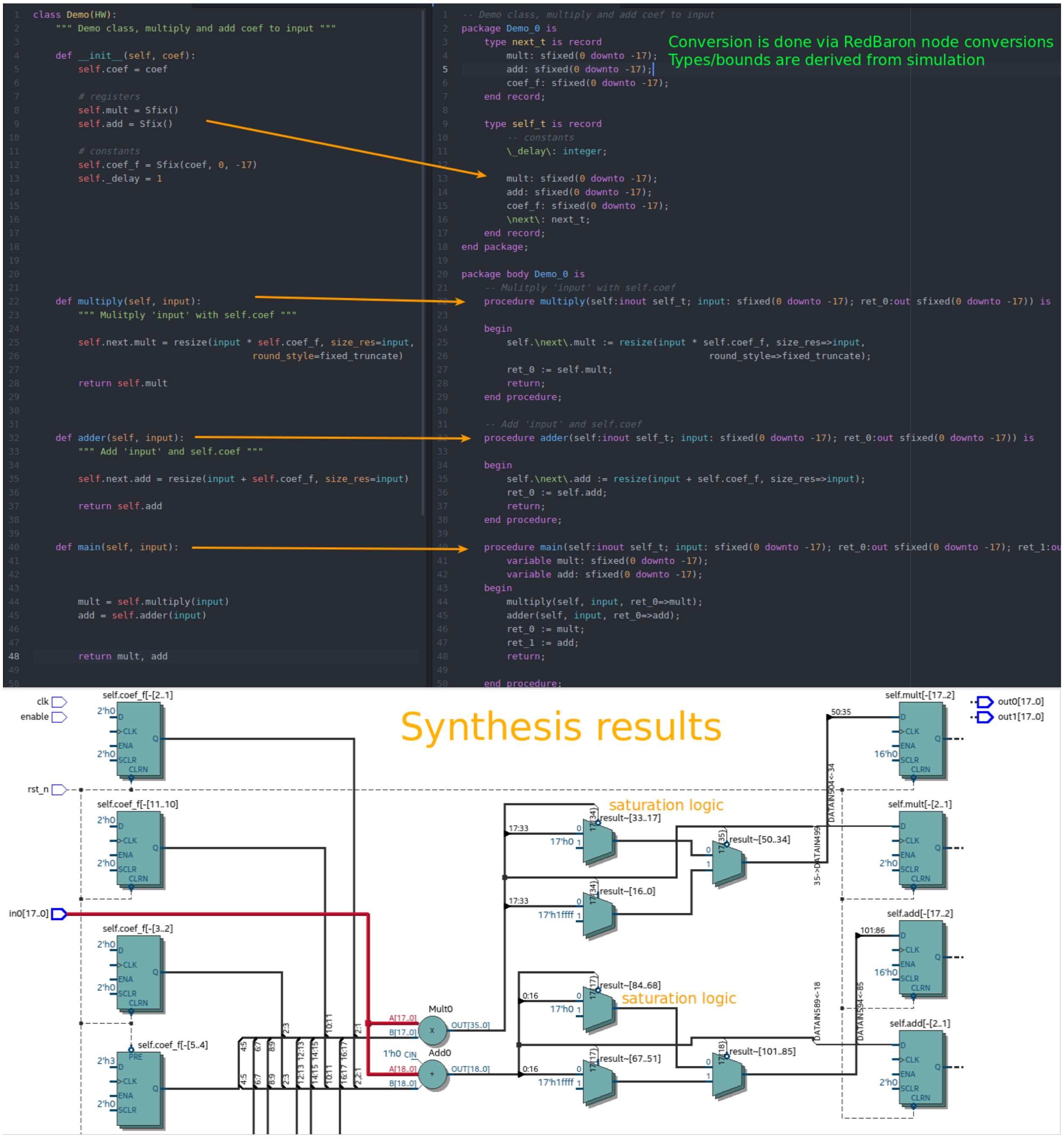

Fig. 1.1: Caption

Limitations/future work

Currently designs are limited to one clock signal, decimators are possible by using Streaming interface. Future plans is to add support for multirate signal processing, this would involve automatic PLL configuration. I am thinking about integration with Qsys to handle all the nasty clocking stuff.

Synthesizability has been tested on Intel Quartus software and on Cyclone IV device (one on BladeRF and LimeSDR). I assume it will work on other Intel FPGAs as well, no guarantees.

Fixed point conversion must be done by hand, however Pyha can keep track of all class and local variables during the simulations, so automatic conversion is very much possible in the future.

Integration to bus structures is another item in the wish-list. Streaming blocks already exist in very basic form. Ideally AvalonMM like buses should be supported, with automatic HAL generation, that would allow design of reconfigurable FIR filters for example.

Objective/goal

Provide simpler way of turning DSP blocks to FPGA. Reduce the gap between regular programming and hardware design. Turn GNURadio flowgraphs to FPGA? Model based verification! Why do it? opensource

How far can we go with the oneprocess design? Everyone else uses VHDL as a very low level interface.

Scope

Focus on LimeSDR board and GnuRadio Pothos, frameworks.

Structure

First chapter of this thesis gives an short background about

Chapter 2

Background

Give a short overview of whats up.

Python

Python is a popular programming language which has lately gained big support in the scientific world, especially in the world of machine learning and data science. It has vast support of scientific packages like Numpy for matrix math or Scipy for scientific computing in addition it has many superb plotting libraries. Many people see Python scientific stack as a better and free MATLAB.

Free Dev tools. .. http://www.scipy-lectures.org/intro/intro.html#why-python

> %https://github.com/jrjohansson/scientific-python-lectures/blob/master/Lecture-0-Scientific-Computing-with-Python.ipynb

HDL related tools in Python

MyHDL

Migen

CocoTb

Chapter 3

Pyha

This paragraph gives an basic overview of the developed tool.

Basics

Pyha extends the VHDL language by allowing objective-oriented designs. Unit object is Python class as shown on

Listing 3.1: Basic Pyha unit

```
class PyhaUnit(HW):
    def __init__(self, coef):
        pass

    def main(self, input):
        pass

    def model_main(self, input_list):
        pass
```

Listing 3.1 shows the besic design unit of the developend tool, it is a standard Python class, that is derived from a baseclass *HW, purpos of this baseclass is to do some metaclass stuff and register this class as Pyha module.

Metaclass actions:

Combinatory logic

Todo

Ref comb logic.

Listing 3.2: Basic combinatory circuit in Pyha

```
class Comb(HW):
    def main(self, a, b):
        xor_out = a xor b
        return xor_out
```

Listing 3.2 shows the design of a combinatory logic. In this case it is a simple xor operation between two input operands. It is a standard Python class, that is derived from a baseclass *HW, purpose of the baseclass is to do some metaclass stuff and register this class as Pyha module.

Class contains an function 'main', that is considered as the top level function for all Pyha designs. This function performs the xor between two inputs 'a' and 'b' and then returns the result.

In general all assigments to local variables are interpreted as combinatory logic.

Todo

how this turns to VHDL and RTL picture?

Sequential logic

Todo

Ref comb logic.

Listing 3.3: Basic sequential circuit in Pyha

```
class Reg(HW):
    def __init__(self):
        self.reg = 0

    def main(self, a, b):
        self.next.reg = a + b
        return self.reg
```

Listing 3.3 shows the design of a registered adder.

In Pyha, registers are inferred from the ogject storage, that is everything defined in 'self' will be made registers.

The 'main' function performs addition between two inputs 'a' and 'b' and then returns the result. It can be noted that the sum is assigned to 'self.next' indicating that this is the next value register takes on next clock.

Also returned is self.reg, that is the current value of the register.

In general this system is similiar to VHDL signals:

- Reading of the signal returns the old value
- Register takes the next value in next clock cycle (that is self.next.reg becomes self.reg)
- Last value written to register dominates the next value

However there is one huge difference aswell, namely that VHDL signals do not have order, while all Pyha code is stctural.

Todo

how this turns to VHDL and RTL picture?

Types

This chapter gives overview of types supported by Pyha.

Integers

Integer types and operations are supported for FPGA conversion with a couple of limitations. First of all, Python integers have unlimited precision *[8]*. This requirement is impossible to meet and because of this converted integers are assumed to be 32 bits wide.

Conversion wize, all inger objectsa are mapped to VHDL type 'integer', that implements 32 bit signed integer. In case integer object is returned to top-module, it is converted to 'std_logic_vector(31 downto 0)'.

Booleans

Booleans in Python are truth values that can either be True or False. Booleans are fully supported for conversion. In VHDL type 'boolean' is used. In case of top-module, it is converted to 'std_logic' type.

Floats

Floating point values can be synthesized as constants only if they find a way to become fixed_point type. Generally Pyha does not support converting floating point values, however this could be useful because floating point values can very much be used in RTL simulation, it could be used to verify design before fixed point conversion.

Floats can be used as constants only, in coperation with Fixed point class.

Fixed-point type

Fixed point numbers can be to effectively turn floating point models into FPGA.

Todo

ref http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.129.5579&rep=rep1&type=pdf https://www.dsprelated.com/showarticle/139.php

Fixed point numbers are defined to have bits for integer size and fractional size. Integer bits determine the maximum size of the number. Fractional bits determine the minimum resolution.

Main type of Pyha is Sfix, that is an signed fixed point number.

```
>>> Sfix(0.123, left=0, right=-17)
0.1230010986328125 [0:-17]
>>> Sfix(0.123, left=0, right=-7)
0.125 [0:-7]
```

Overflows and Saturation

Practical fixed-point variables can store only a part of what floating point value could. Converting a design from floatin to fixed point opens up a possiblity of overflows. That is, when the value grows bigger or smaller than the format can reprsent. This condition is known as overflow.

By default Pyha uses fixed-point numbers that have saturaton enabled, meaning that if value goes over maximum possible value, it is instead kept at the maximum value. Some examples:

```
>>> Sfix(2.5, left=0, right=-17)
WARNING:pyha.common.sfix:Saturation 2.5 -> 0.9999923706054688
0.9999923706054688 [0:-17]
>>> Sfix(2.5, left=1, right=-17)
```

```
WARNING:pyha.common.sfix:Saturation 2.5 -> 1.9999923706054688
1.9999923706054688 [1:-17]
>>> Sfix(2.5, left=2, right=-17)
2.5 [2:-17]
```

On the other hand, sometimes overflow can be a feature. For example, when designing free running counters. For this usages, saturation can be disabled.

```
>>> Sfix(0.9, left=0, right=-17, overflow_style=fixed_wrap)
0.9000015258789062 [0:-17]
```

```
>>> Sfix(0.9 + 0.1, left=0, right=-17, overflow_style=fixed_wrap)
-1.0 [0:-17]
```

Rounding

Pyha support rounding on arithmetic, basically it should be turned off as it costs alot.

Fixed-point arithmetic and sizing rules

Arithmetic operations can be run on fixed point variables as usual. Division is not defined as it is almost always unneccesary in hardware.

Library comes with sizing rules in order to guarantee that fixed point operations never overflow.

For example consider an fixed point number with format that can represent numbers betwe

```
>>> Sfix(0.9, 0, -17)
0.9000015258789062 [0:-17]
```

Now adding two such numbers:

```
>>> Sfix(0.9, 0, -17) + Sfix(0.9, 0, -17)
1.8000030517578125 [1:-17]
```

While this operation should overflow, it did not. Because fixed point library always resizes the output for the worst case. In case of addition it always adds one integer bit to accumulate possible overflows.

But note that this system is not very smart, if we would add up such numbers 100 times, it would add 100 bits to the integer portion of the number.

The philosophy of fixed point library is to guarantee no precision loss happens during arithmetic operations, in order to do this it has to extend the output format. It is designers job to resize numbers back into optimal format after operations.

Resizing

Fixed point number can be forced to whatever size by using the resize functionality.

```
>>> a = Sfix(0.89, left=0, right=-17)
>>> a
0.8899993896484375 [0:-17]
>>> b = resize(a, 0, -6)
>>> b
0.890625 [0:-6]
```

```
>>> c = resize(a, size_res=b)
>>> c
0.890625 [0:-6]
```

Pyha support automatic resizing for registers. All assignments to registers will be automatically resized to the original type of the definition.

Conversion to VHDL

VHDL comes with a strong support for fixed-point types by providing and fixed point package in the standard library. More information is about this package is given in *[7]*.

In general Sfix type is built in such a way that all the functions map to the VHDL library, so no conversion is neccesary.

Another option would have been to implement fixed point compiler on my own, it would provide more flexibility but it would take many time + it has t be kept in mind that the VHDL library is already production-tested. Ths mapping to VHDL library seemed like the best option.

It limits the conversion to VHDL only, for example Verilog has no fixed point package in standard library.

Complex fixed-point

Objective of this tool was to simplify model based design and verification of DSP to FPGA models. One frequent problem with DSP models was that they commonly want to use complex numbers. In order to unify the interface of the model and hardware model, Pyha supports complex numbers for interfacing means, arithmetic operations are not defined. That means complex values can be passed arond and registered but arithmetics must be done on `.real` and `.imag` elements, that are just Sfix objects.

```
>>> a = ComplexSfix(0.45 + 0.88j, left=0, right=-17)
>>> a
0.45+0.88j [0:-17]
>>> a.real
0.4499969482421875 [0:-17]
>>> a.imag
0.8799972534179688 [0:-17]
```

Another way to construct it:

```
>>> a = Sfix(-0.5, 0, -17)
>>> b = Sfix(0.5, 0, -17)
>>> ComplexSfix(a, b)
-0.50+0.50j [0:-17]
```

User defined types / Submodules

For design reuse it is needed to reuse previously generated designs. Traditional HDLs use entity declarations for this purpose. One of the key assumption of these entities is that they all run in parallel. This has some advantages and disadvantages. Good thing is that this is the most flexible solution, that is it supports as many clocks and clock domains as neccessary. Disadvantage is that in the end much of the VHDL programing comes down to wiring togather different entities, and this can be worksome and bugful process.

Another downside is that all of these entities must be simulated as a separate process, this has a cost on simulation speed and more severily it makes debugging hard..think about debugging multi-threaded programs.

In contrast to traditional HDLs, Pyha has taken an approach where design reuse is archived trough regular objects. This has numerous advantages:

- Defining a module is as easy as making an class object
- Using submodule is as easy as in traditional programming..just call the functions
- Execution in same domain, one process design

Result of this design decision is that using submodules is basically the same as in normal programming. This decision comes with a severe penalty aswell, namely all the submodules then must work with the same clock signal. This essentially limits Pyha designs down to using only one clock. This is a serious constrain for real life systems, but for now it can be lived with.

It is possible to get around this by using clock domain crossing interfacec between two Pyha modules.

Support for VHDl conversion is straightforward, as Pyha modules are converted into VHDL struct. So having a submodule means just having a struct member of that module.

Lists

All the previously mentioned convertible types can be also used in a list form. Matching term in VHDL vocabulary is array. The difference is that Python lists dont have a size limit, while VHDL arrays must be always constrained. This is actually not a big problem as the final list size is already known.

VHDL being an very strictly typed language requires an definition of each array type.

For example writing `l = [1, 2]` in Python would trigger the code shown in Listing 3.4, where line 1 is a new array type definitiaon and a second line defines a variable `a` of this type. Note that the elements type is deduced from the type of first element in Python array the size of defined array is as `len(l)-1`.

Listing 3.4: VHDL conversion for integer array

```
1 type integer_list_t is array (natural range <>) of integer;
2 l: integer_list_t(0 to 1);
```

Chapter 4

Conversion

This chapter examines the feasability and means of converting Python code to VHDL.

What about verilog?

Python vs VHDL

VHDL is known as a strongly typed language in addition to that it is very verbose. Python is dynamically typed and is basically as least verbose as possible.

Comparison of syntax

Assignments

In VHDL

The syntax of a variable assignment statement is `variable-name := value-expression;` The immediate assignment notion, :=, is used for the variable assignment. There is no time dimension (i.e., no propagation delay) and the assignment takes effect immediately. The behavior of the variable assignment is just like that of a regular variable assignment used in a traditional programming language. *[1]*

The syntax of a sequential signal assignment is identical to that of the simple concurrent signal assignment of Chapter 4 except that the former is inside a process. It can be written as signal-name <= projected-waveform; The projected-waveform clause consists of a value expression and a time expression, which is generally used to represent the propagation delay. As in the concurrent signal assignment statement, the delay specification cannot be synthesized and we always use the default &delay. The syntax becomes signal-name <=

value-expression; Note that the concurrent conditional and selected signal assignment statements cannot be used inside the process. For a signal assignment with 6-delay, the behavior of a sequential signal assignment statement is somewhat different from that of its concurrent counterpart. If a process has a sensitivity list, the execution of sequential statements is treated as a "single abstract evaluation," and the actual value of an expression will not be assigned to a signal until the end of the process. This is consistent with the black box interpretation of the process; that is, the entire process is treated as one indivisible circuit part, and the signal is assigned a value only after the completion of all sequential statements. Inside a process, a signal can be assigned multiple times. If all assignments are with &delays, only the last assignment takes effect. Because the signal is not updated until the end of the process, it never assumes any "intermediate" value. For example, consider the following code segment: *[1]*

Python support

Supporting VHDL variable assignment in Python code is trivial, only the VHDl assignment notation must be changed from `:=` to `=`.

Pyhas solution simplifies the VHDL assignments by have unified style with still same functionality.

Support for VHDl simulation needs to after the clock tick update the next values into actual values.

Design resuse

Object-orientation support

Major goal of this project is to support object-oriented hardware design.

Goal is to provide simple object support, advanced features like inherintance and overloadings are not considerted at this moment.

Python itself comes with a strong object-orientation support. On the other hand VHDL has no class support whatsoever.

Listing 4.1: Basic class in Python

```
class Name:
    def __init__(self):
        self.instance_member = 0

    def function(self, a, b):
```

```
        self.instance_member = a + b
        return self.instance_member
```

Listing 1.1 shows an simple example of Python class. It has two functions, `__init__` in python is a class constructor. `function` is just and user defined function.

It can be used as follows:

```
>>> a = Name()
>>> a.instance_member
0
>>> a.function(1, 2)
3
>>> a.instance_member
3
```

Turning this kind of structure to VHDL can be done by levraging VHDL support for struct types.

Listing 4.2: VHDL conversion for integer array

```
type self_t is record
    instance_member: integer;
end record;

procedure main(self:inout self_t; a: integer; ret_0:out integer) is
begin
    self.instance_member := a;
    ret_0 := self.instance_member;
    return;
end procedure;
```

Convertings

Based on the results of previous chapter it is clear that specific Python code can be converted to VHDL. Doing so requires some way of parsing the Python code and outputting VHDL.

In general this step involves using an abstract syntax tree (AST). MyHDL is using this solution.

However RedBaron offers a better solution. RedBaron is an Python library with an aim to significally simply operations with source code parsing. Also it is not based on the AST, but on FST, that is full syntax tree keeping all the comments and stuff.

Here is a simple example:

```
>>> red = RedBaron('a = b')
>>> red
0   a = b
```

RedBaron turns all the blocks in the code into special 'nodes'. Help function provides an ex

```
>>> red.help()
0 -----------------------------------------------------
AssignmentNode()
  # identifiers: assign, assignment, assignment_, assignmentnode
  operator=''
  target ->
    NameNode()
      # identifiers: name, name_, namenode
      value='a'
  value ->
    NameNode()
      # identifiers: name, name_, namenode
      value='b'
```

Now Pyha defined a mirror node for each of RedBaron nodes, with the goal of turning the code into VHDL. For example in the above example main node is AssignmentNode, this could be modified to change the '=' into ':=' and add ';' to the end of line. Resulting in a VHDL compatible statement:

```
a := b;
```

Converting functions

First of all, all the convertable functions are assumed to be class functions, that means they have the first argument `self`.

Python is very liberal in syntax rules, for example functions and even classes can be defined inside functions. In this work we focus on functons that dont contain these advanced features.

VHDL supports two style of functions:

- Functions - classical functions, that have input values and can return one value
- Procedures - these cannot return a value, but can have agument that is of type 'out', thus returing trough an

output argument. Also it allows argument to be of type 'inout' that is perfect for class object.

All the Python functions are to be converted to VHDL procedures as they provide more wider interface.

Python functions can return multiple values and define local variables. In order to support multiple return, multiple output arguments are appended to the argument list with prefix `ret_`. So for example first return would be assigned to `ret_0` and the second one to `ret_1`.

Here is an simple Python function that contains most of the features required by conversion, these are:

- First argument self
- Input argument
- Local variables
- Multiple return values

```
def main(self, a):
    b = a
    return a, b
```

Listing 4.3: VHDL example procedure

```
procedure main(self:inout self_t; a: integer; ret_0:out integer; ret_1:out integer) is
    variable b: integer;
begin
    b := a;
    ret_0 := a;
    ret_1 := b;
    return;
end procedure;
```

In VHDL local variables must be defined in a special region before the procedure body. Converter can handle these caese thanks to the previously discussed types stuff.

The fact that Python functions can return into multiple variables requires and conversion on VHDL side:

```
ret0, ret1 = self.main(b)
```

```
main(self, b, ret_0=>ret0, ret_1=>ret1);
```

Problem of types

Biggest difference and problem between Python and VHDL is the type system. While in VHDL everything must be typed, Python is fully dynamically typed language, meaning that types only come into play when the code is executing.

In general there are some different approaches to solve this problem:

- Determining types from Python source code
- Determining types from one pass execution/initial execution
- Using longer simulation

First option is attractive as it could convert without any side actions, problem with this approach is that the converter would have to be extreamly complex in order to infer the variable types. For example `a = 5` is a simple example that type is integer, but for example `a = b` type is not clear. Converter would have to look up the type of b, but which b? in which scope? etc. It is clear that this solution is not reasonable to solve.

Second option would use the result of initial execution of classes. In python defining an class object automatically executes its constructor(`def __init__(self)`). Basically theis would allow to determine all the class variables types, by just making the object. It would be as good as the first option really, but simplifies the type deduction significaly. Still type info provided here is not enough, for example local variables are not covered. One way would e to use only class variables, but this has slight downsides aswell.

Last option would simulate the whole design in order to figure out every type in the design. After each execution to the function, latest call stack is preserved (this includes all the values of locals). PyPy also uses system like this. Downside of this solution is obviously that the desing must be simulated in Python domain before it can be converted to VHDL.

Also the simulation data must cover all the cases, for example consider the function with conditional local variable, as shown on Listing 4.4. If the simulaton passes only True values to the function, value of variable 'b' will be unknown ad vice-versa. This is a problem but not a huge one because in hardware...

Listing 4.4: Type problems

```
def main(c):
    if c:
        a = 0
    else:
        b = False
```

Other advantages this way makes possible to use 'lazy' coding, meaning that only the type after the end of simulation matters.

Language differences...

Extensions..wehn you can do more in python domain.

Feasability of converting Python to VHDL

Simulation and verification

Make separate chapter for testing and verification? Basics can be described here. Requrements...want RTL sim, GATE sim, in loop etc

Implementation of the simulation code relies heavily on the signal assignment semantics. Basically code writes to the 'next' element and thats it. After the top-level function call, all the 'next' values must be propagated into the original registers. This process is basically an clock tick

Essentially this comes downt to being and VHDL simulator inside VHDL simulator. it may sound stupid, but it works for simulations and synthesys, so i guess it is not stupid.

Python simulation

RTL simulation

Testing

Chapter 5

Design flow

This chapter aims to investigate how modern software development techniques coulde be used in design of hardware.

While MyHDL brings development to the Python world, it still requires the make of test-benches and stuff. Pyha aimst to simplify this by providing higl level simulation functions.

Convetional design flow

VHDL uuendused? VUNIT VUEM?

Test-driven development / unit-tests

Model based development

How MyHDl and other stuffs contribute here?

Pyha support

Since Pyha brings the development into Python domain, it opens this whole ecosystem for writing testing code.

Python ships with many unit-test libraries, for example PyTest, that is the main one used for Pyha.

As far as what goes for model writing, Python comes with extensive schinetific stuff. For example Scipy and Numpy. In addition all the GNURadio blocks have Python mappings.

Simplifying testing

One problem for model based designs is that the model is generally written in some higher level language and so testing the model needs to have different tests than HDL testing. That is one ov the problems with CocoTB.

Pyha simplifies this by providing an one function that can repeat the test on model, hardware-model, RTL and GATE level simulations.

Ipython notebook

It is interactive environment for python. Show how this can be used.

Chapter 6

Design examples

This chapter provides some example designs implemented using the experimental compiler.

First example developes and moving-average filter.

First three examples will interatively implement DC-removal system. First design implements an simple fixed-point accumulator. Second one builds upon this and implements moving average filter. Lastly multiple moving average filters are chained to form a DC removal circuit.

Second example is an FIR filter, with reloadable switchable taps ?

Third design example shows how to chain togather already exsisting Pyha blocks to implement greater systems. In this case it is FSK receiver. This examples does not go into details.

Moving Average

The moving average is the most common filter in DSP, mainly because it is the easiest digital filter to understand and use. In spite of its simplicity, the moving average filter is optimal for a common task: reducing random noise while retaining a sharp step response. This makes it the premier filter for time domain encoded signals. However, the moving average is the worst filter for frequency domain encoded signals, with little ability to separate one band of frequencies from another. Relatives of the moving average filter include the Gaussian, Blackman, and multiple- pass moving average. These have slightly better performance in the frequency domain, at the expense of increased computation time. *[2]*

Consider following data:

```
>>> l = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6]
>>> out[0] = (l[0] + l[1]) / 2
>>> out[1] = (l[1] + l[2]) / 2
>>> out[2] = (l[2] + l[3]) / 2
```

Somehow explain how this stuff is equal to convolution.

Listing 6.1: Implementation of moving average algorithm in Python

```
avg_len = 4
taps = [1 / avg_len] * avg_len
ret = np.convolve(inputs, taps, mode='full')
```

Listing 6.1 shows how to implement moving average algorithm in Python, it uses the fact that it is basically convolution...bla bla bla.

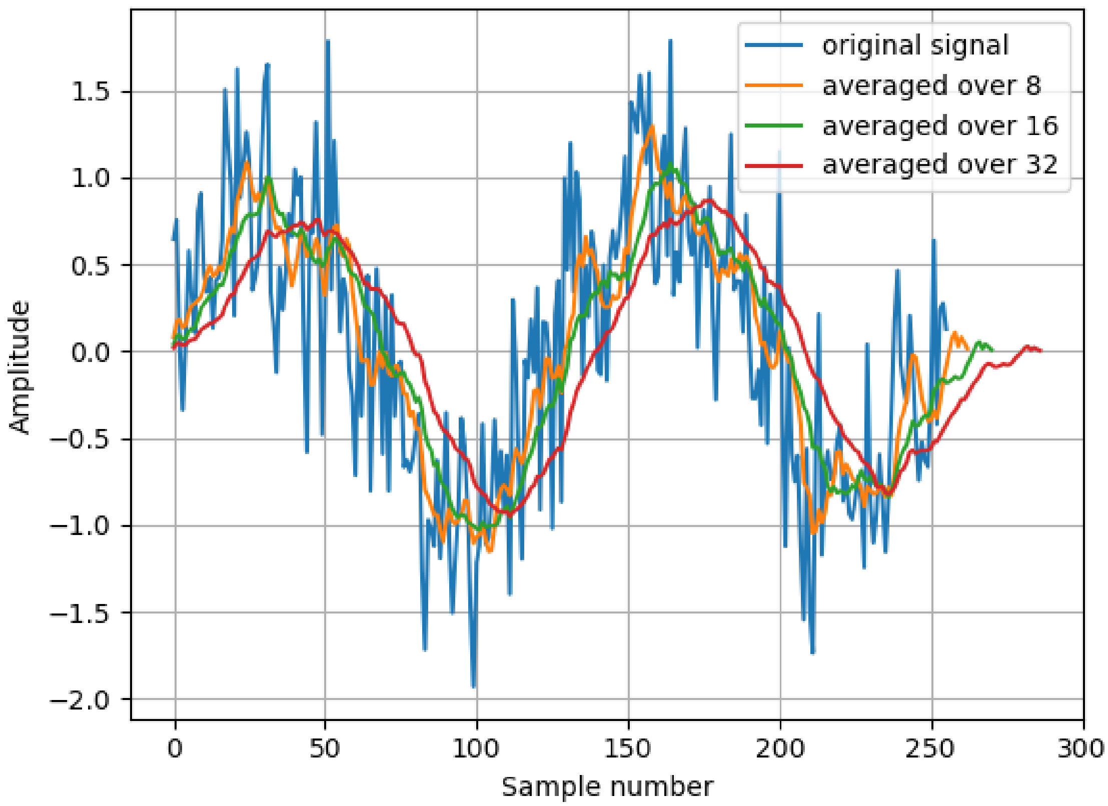

Fig. 6.1: Example of moving averager as noise reduction

As shown on Fig. 6.1, moving average is a good noise reduction algorithm. Increasing the averaging window reduces more noise but also increases the complexity and delay of the system.

In addition, moving average is also an optimal solution for performing matched filtering of rectangular pulses *[2]*. On Fig. 6.2 (a) digital signal is corrupted with noise, by using moving average with length equal to the signal samples per symbol, enables to recover the signal and send it to sampler (b).

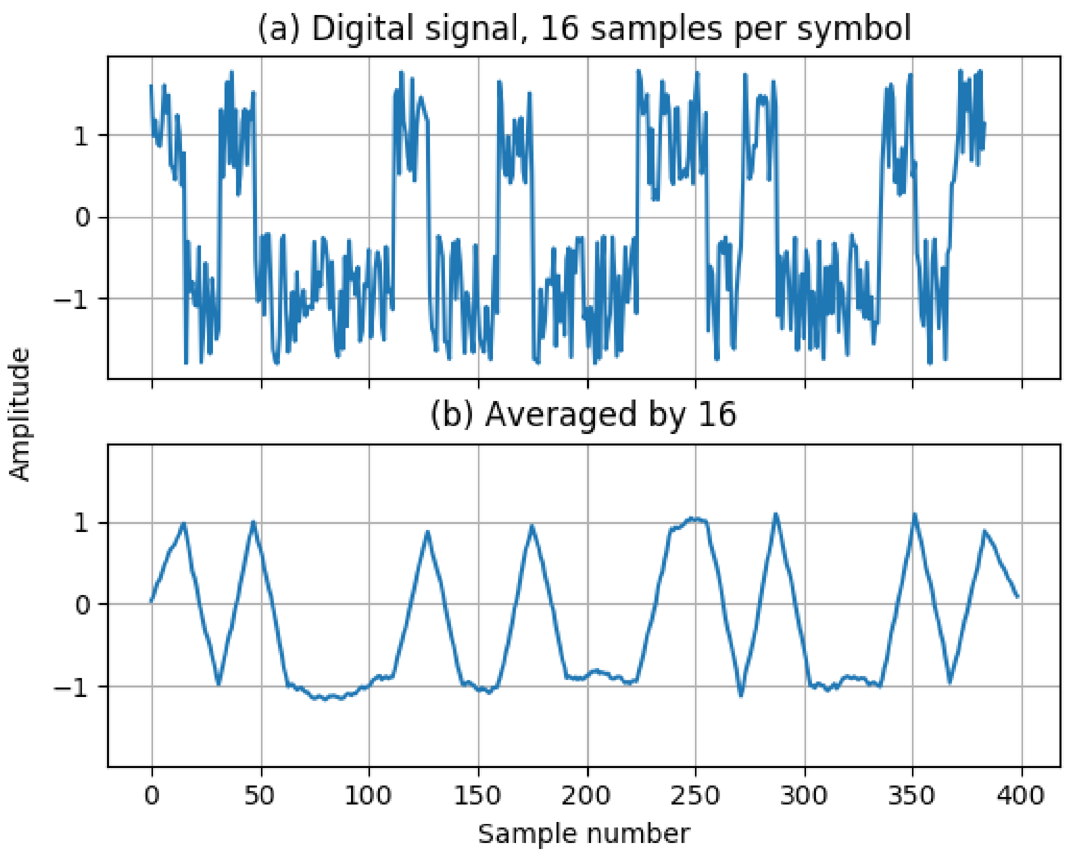

Fig. 6.2: Moving average as matched filter

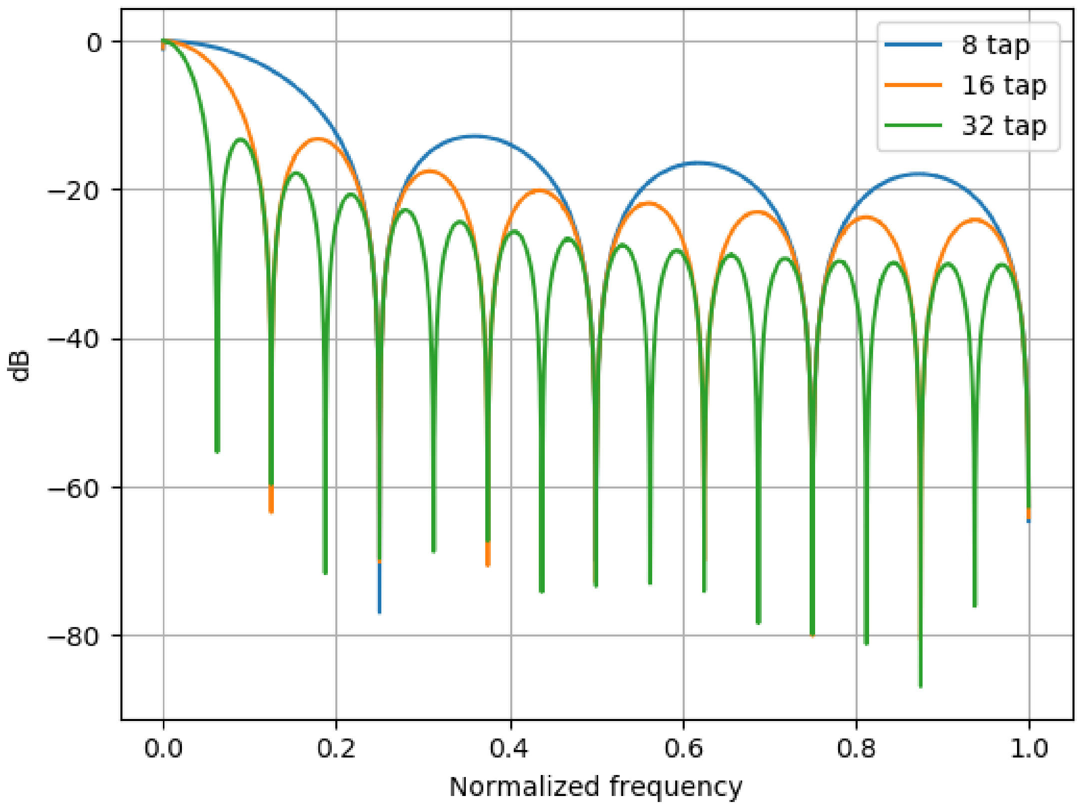

Fig. 6.3: Frequency response of moving average filter

Fig. 6.3 shows that the moving average algorithm acts basically as a low-pass filter in the frequency domain. Passband width and stopband attenuation are controlled by the moving averages length. Note that when taps number get high, then moving average basically returns the DC offset of a signal.

In short, the moving average is an exceptionally good smoothing filter (the action in the time domain), but an exceptionally bad low-pass filter (the action in the frequency domain). *[2]*

Implementing the model

As shown in the previous chapter, in Pyha, model can be one part of the class definition. This helps to keep stuff synced.

Listing 6.2: Moving average model and tests

```
class MovingAverage(HW):
    def __init__(self, window_len):
        self.window_len = window_len

    def model_main(self, inputs):
        taps = [1 / self.window_len] * self.window_len
        ret = np.convolve(inputs, taps, mode='full')
        return ret[:-self.window_len + 1]

def test_basic():
    mov = MovingAverage(window_len=4)
    x = [-0.2, 0.05, 1.0, -0.9571, 0.0987]
    expected = [-0.05, -0.0375, 0.2125, -0.026775, 0.0479]
    assert_sim_match(mov, expected, x, simulations=[SIM_MODEL])

def test_max():
    mov = MovingAverage(window_len=4)
    x = [1., 1., 1., 1., 1., 1.]
    expected = [0.25, 0.5, 0.75, 1., 1., 1.]
    assert_sim_match(mov, expected, x, simulations=[SIM_MODEL])
```

Listing 6.2 defines an `MovingAverage` class which includes the special `model_main` function, dedicated for defining model code. In addtion it defines 2 simple tests, in general there should be more tests defined but here we keep things minimal.

`test_max` tests the model for maximum valued inputs, assuming that we are working with numbers that are normalized to [-1, 1] range. `test_basic` uses just some random data and expected output.

Implementing for hardware

Hardware implementation of moving average could be to implement a convolution, but this takes alot of resources and frankly is an overkill.

A tremendous advantage of the moving average filter is that it can be implemented with an algorithm that is very fast. To understand this algorithm, imagine passing an input signal, , through a seven point moving x[] average filter to form an output signal, . Now look at how two adjacent y[] output points, and , are calculated:

```
>>> y[4] = x[1] + x[2] + x[3] + x[4]
>>> y[5] = x[2] + x[3] + x[4] + x[5]
>>> y[6] = x[3] + x[4] + x[5] + x[6]
```

These are nearly the same calculation. If `y[4]` has already been calculated, the most efficient way to calculate `y[5]` is:

```
>>> y[5] = y[5] + x[5] + x[1]
:cite:`dspbook`
```

Listing 6.3: Moving average hw model

```
# THIS CODE IS SHIT
class MovingAverage(HW):
    def __init__(self, window_len):
        self.window_pow = int(np.log2(window_len))

        # registers
        self.shift_register = [Sfix()] * self.window_len
        self.sum = Sfix(left=self.window_pow, overflow_style=fixed_wrap, round_
↪style=fixed_truncate)

        # module delay
        self._delay = 1

    def main(self, x):
        # add new element to shift register
        self.next.shift_register = [x] + self.shift_register[:-1]

        # calculate new sum
        self.next.sum = self.sum + x - self.shift_register[-1]

        # divide sum by amount of window_len, and resize to same format as input
↪'x'
        ret = resize(self.sum >> self.window_pow, size_res=x)
        return ret
```

```
    def model_main(self, inputs):
        ...
```

In order to implement this in hardware we must define some registers. First we need to keep track of last `window_len` inputs, for that the standard way is to write a shift register. Shift register is basically just an fixed size array that on each clock tick takes in a new values and shifts out the oldest value (to make space for the new one).

Secondary we need to keep track of the sum. Since this is an accumulator, we need to provide a large enough integer side to avoid overflows. As we know the `window_len` and that the input numbers are normalized we can calculate that the maximum value this sum can take is infact equal to `window_len`. Then we use the bit counts as left value to avoid overflows in the core.

Also due to the registers in the signal path we have to specify it, by using `self._delay` . Since we added two registers we set this to value 2.

Testing the newly written code is very simple, we just have to add required simulation flags to the already written unit tests.

Conversion and RTL simulations

Conversion is done as a part of running the unit-test with `SIM_RTL` mode.

Listing 6.4: Main function of converted VHDL sources

```
procedure main(self:inout self_t; x: sfixed(0 downto -17); ret_0:out sfixed(0␣
↪downto -17)) is

begin

    -- add new element to shift register
    self.\next\.shift_register := x & self.shift_register(0 to self.shift_
↪register'high-1);

    -- calculate new sum
    self.\next\.sum := resize(self.sum + x - self.shift_register(self.shift_
↪register'length-1), 2, -17, fixed_wrap, fixed_truncate);

    -- divide sum by amount of window_len
    self.\next\.\out\ := resize(self.sum sra self.window_pow, 0, -17, fixed_wrap,
↪ fixed_truncate);
    ret_0 := self.\out\;
    return;
end procedure;
```

Listing 6.4 shows the significant part of the conversion process. As seen it looks very similiar to the Python function. Full output of the conversion is can be seen at repo[1].

GATE level simulation

As written in some chapter, Pyha supports also rupports running GATE-level simulations by integrating with Intel Quartus software

Running the GATE simulation, will produce 'quartus' directory in dir_path. One useful tool in Quartus software is RTL viewer, it can be opened from Tools-Netlist viewers-RTL viewer.

RTL of this tutorial:

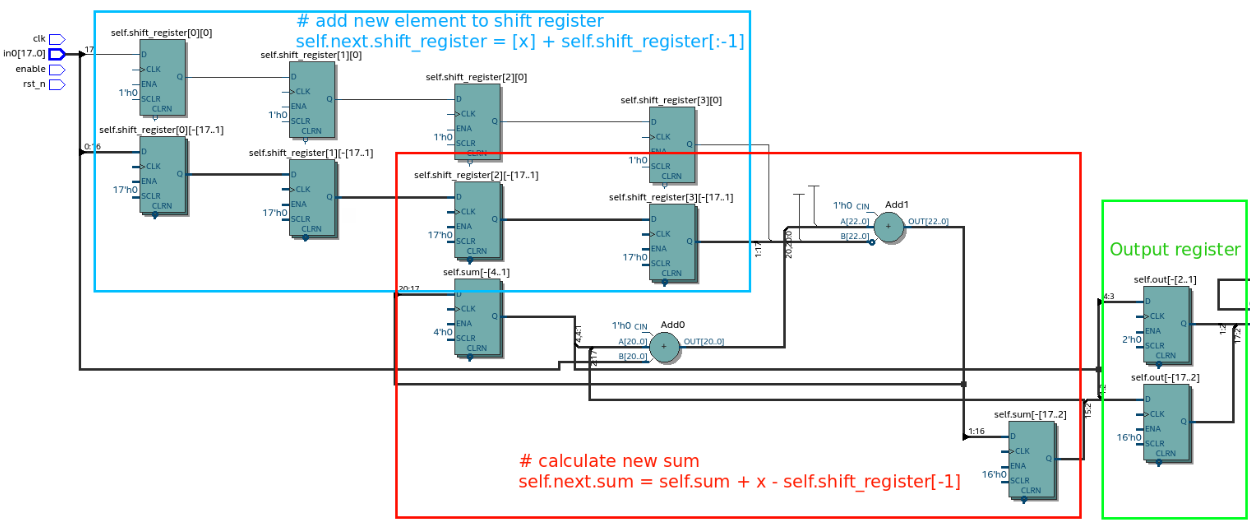

Fig. 6.4: RTL view of moving average (Intel Quartus RTL viewer)

Fig. 1.1 shows the synthesized result of this work. The blue box shows the part of the logic that was inferred as to be shift register, red part contains all the logic, as expected two adders are requires. Finally green part is the output register.

Quartus project can be seen at repo[1].

Resource usage

All the synthesis tests are to be run on the EP4CE40F23C8N chip by Altera. It is from Cyclone IV family. In todays standard this is quite an mediocer chip, behind two generations. It was chosen because BladeRF and LimeSDR use this chip. It costs about 60 euros (Mouser)

Some features of this FPGA *[3]*:

[1] https://github.com/petspats/thesis/tree/master/examples/moving_average/conversion

- 39,600 logic elements
- 1,134Kbits embedded memory
- 116 embedded 18x18 multipliers
- 4 PLLs

Synhesizing with Quartus gave following resorce usage:

- Total logic elements: 94 / 39,600 (< 1 %)
- Total memory bits: 54 / 1,161,216 (< 1 %)
- Embedded multipliers: 0 / 232 (0 %)

In additon, maximum reported clock speed is 222 MHz, that is over the 200 MHz limit of Cyclone IV device *[3]*.

Linear phase DC Removal

Direct conversion (homodyne or zero-IF) receivers have become very popular recently especially in the realm of software defined radio. There are many benefits to direct conversion receivers, but there are also some serious drawbacks, the largest being DC offset and IQ imbalances. DC offset manifests itself as a large spike in the center of the spectrum. This happens in direct conversion receivers due to a few different factors. One is at the ADC where being off by a single LSB will yield a DC offset. Another is at the output of the low-pass filters where any DC bias will propagate through. The last is at the mixer where the local oscillator (LO) being on the center of the desired frequency will leak through to the receiver. *[4]*

In frequency domain, DC offset will look like a peak near the 0 Hz. In time domain, it manifests as a constant component on the hermonic signal.

In *[5]* Rick Lyons investigates the feasability of using moving average algorithm as a DC removal circuit, as shown on Fig. 6.5. This structure has a few problems, first of that it forces to use moving averager with length not power of 2, that would significally complicate the hardware implmenentation.

Second problem is seen on Fig. 6.6. Total ripple of the filter is up to 3 dB, that is 2 times of a difference.

Much better performance can be arcieved by chaining multiple stages of moving averaging, as shown in Fig. 6.7. Chaining them up also helps the power of 2 problem.

New frequency response can be observer on Fig. 6.8. It is clear that the passband ripple has significantly reduced. In addition the cutoff is sharper.

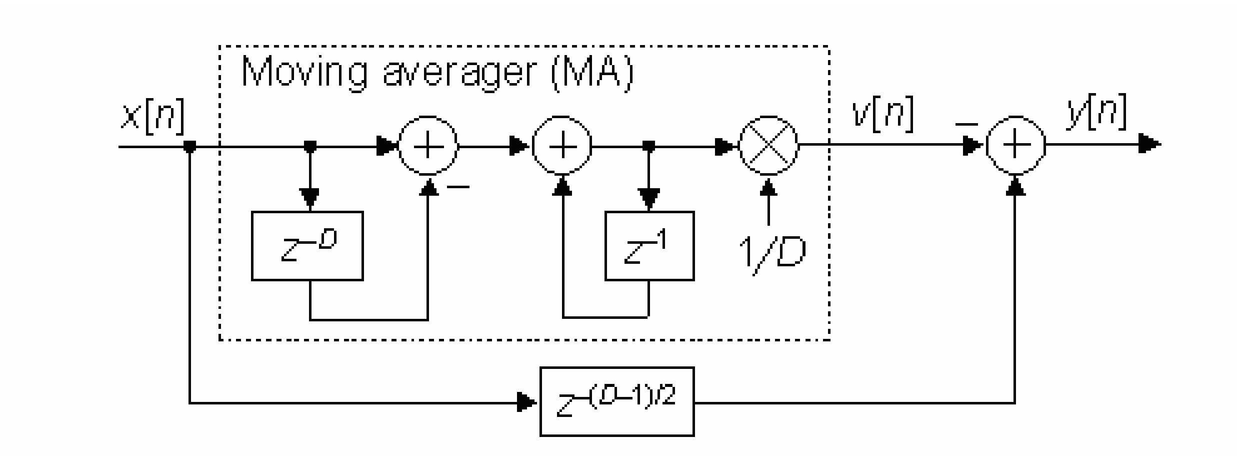

Fig. 6.5: Basic DC removal using moving averager [5]

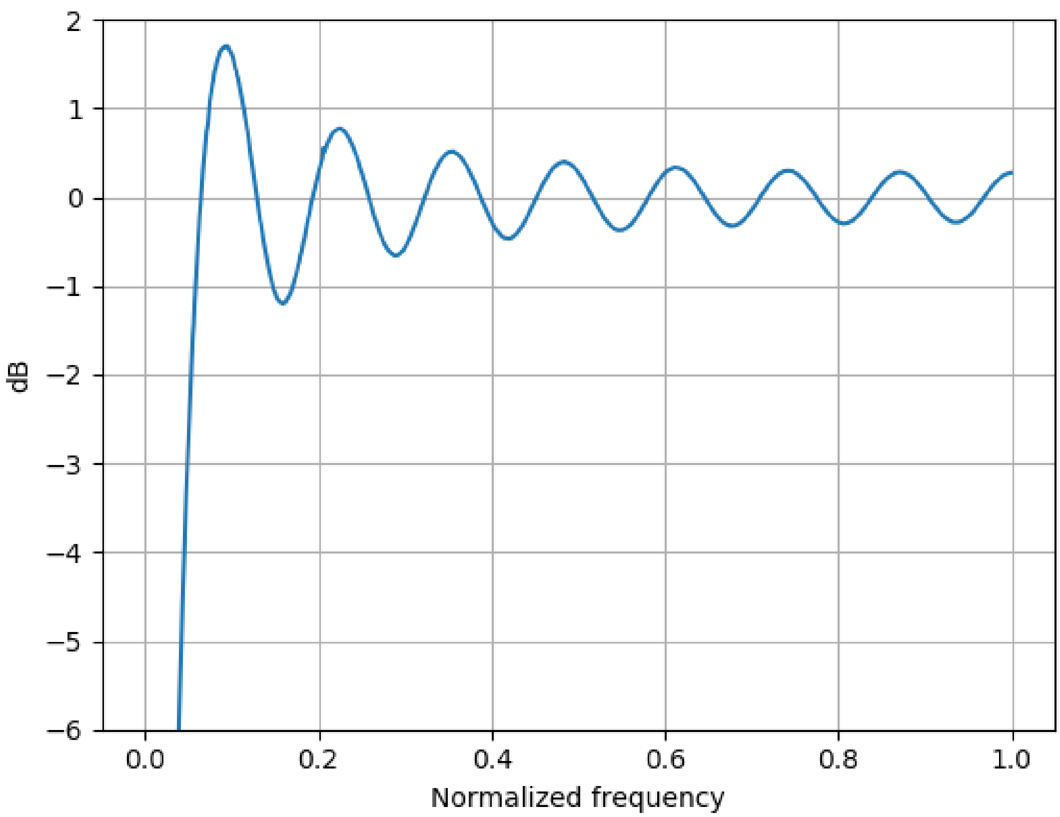

Fig. 6.6: Frequency response of DC removal circuit with Moving average length 31

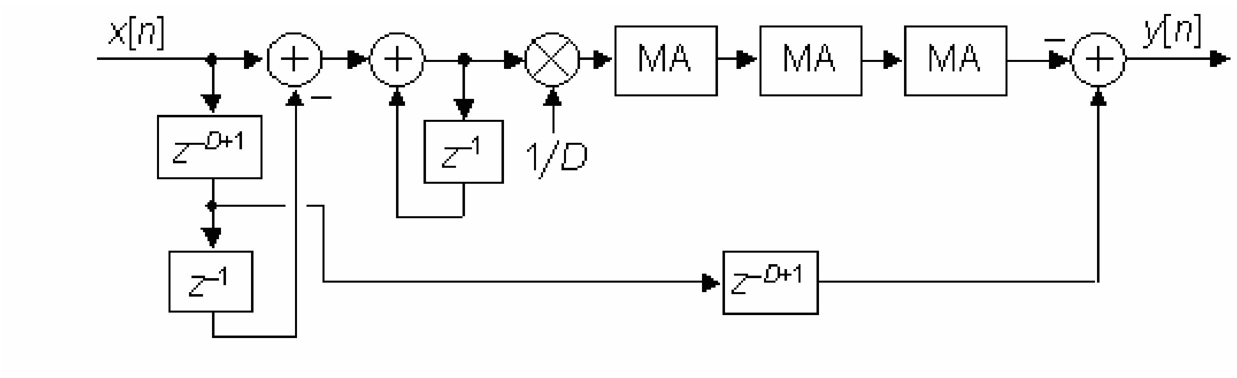

Fig. 6.7: Removing DC with chained moving averagers [5]

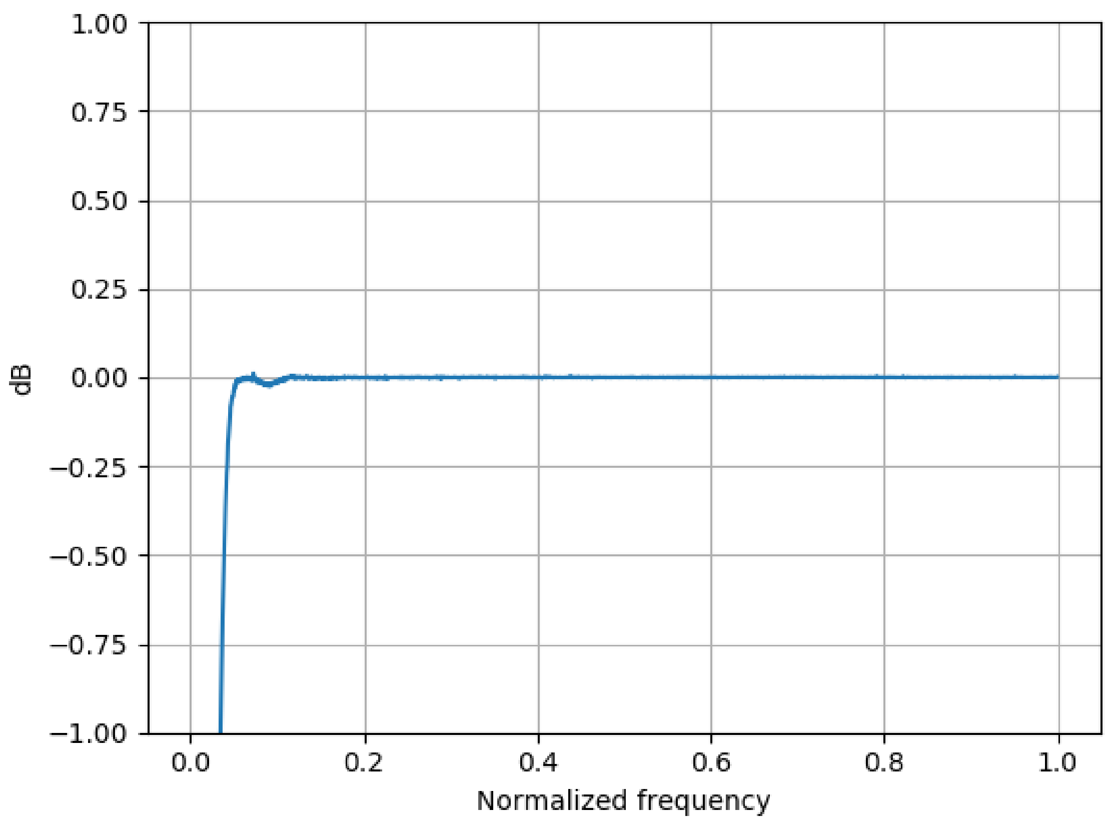

Fig. 6.8: Frequency response of DC removal, 4 cascaded moving averagers

Implementation with Pyha

Implementation is rather straight forward, as shown on Fig. 6.7, algorithm must run input signal over multiple moving average filters (that we have already implemented in previous chapter) and then substract the filter chain output of the delayed input signal.

Listing 6.5: Parametrizable DC-Removal implementation

```
class DCRemoval(HW):
    def __init__(self, window_len, averagers):
        self.mavg = [MovingAverage(window_len) for _ in range(averagers)]

        # this is total delay of moving averages
        hardware_delay = averagers * MovingAverage(window_len)._delay
        self.group_delay = int(averagers * MovingAverage(window_len)._group_
↪delay)
        total_delay = hardware_delay +  self.group_delay

        #registers
        self.input_shr = [Sfix()] * total_delay
        self.out = Sfix(0, 0, -17)

        # module delay
        self._delay = total_delay + 1

    def main(self, x):
        tmp = x
        for mav in self.mavg:
            tmp = mav.main(tmp)

        self.next.input_shr = [x] + self.input_shr[:-1]
        self.next.out = self.input_shr[-1] - tmp
        return self.out

    def model_main(self, x):
        # run signal over all moving averagers
        tmp = x
        for mav in self.mavg:
            tmp = mav.model_main(tmp)

        # subtract from delayed input
        return x[:-self.group_delay] - tmp[self.group_delay:]
```

Listing 6.5 shows the Python implementation. Class is parametrized so that count of moving averagers and the window length can be changed on definiton. Overall it is a pretty straigth forward Python code.

One thing to note that the `model_main` and `main` are nearly identical. That shows that Pyha has archived one of the goals by simplifying hardware design portion.

Unit test for this module have not been listed as most of the testing is done in Ipython Notebook environment, as written in some chapter Pyha is capable or collecting these tests for unit-testing. Can be seen here.

GATE level simulation

As written in some chapter, Pyha supports also rupports running GATE-level simulations by integrating with Intel Quartus software.

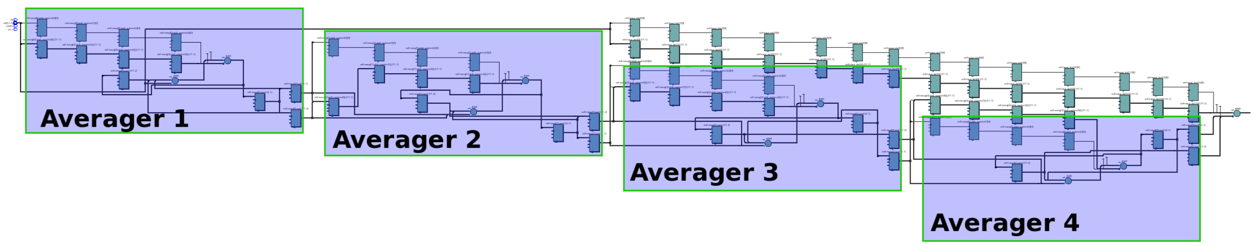

Fig. 6.9: RTL view of simplified DC-Removal (Intel Quartus RTL viewer)

Fig. 6.9 shows an simplified RTL view of the DC removal circuit, it uses averages with length 4 to make RTL plottable. There are 4 averages in total, leftover logic is the delay line and the final substractor.

Quartus project can be seen at repo **[#dcrepo]_**.

Resource usage

Resource usage is returned for the full size circuit, that is 4 chained moving averages with each having 32 taps. Synhesizing with Quartus gave following resource usage:

- Total logic elements: 341 / 39,600 (< 1 %)
- Total memory bits: 2,736 / 1,161,216 (< 1 %)
- Embedded multipliers: 0 / 232 (0 %)

Maximum reported clock speed is 188 MHz (standard compilation).

Conclusions

This chapter showed how to use Pyha to design an efficient, linar phase DC removal circuit. It is clear that making these kind of designs is possible in Pyha and is not significantly harder that coding for the 'model'. Also it showed how design reuse is archieved in Pyha, by reusing Moving average stuff.

Further improvements

Problem with this filter is the delay on the signal path. In this case we used 4 filters with 32 taps, this gives group delay of 64 samples + hardware related delays. Possible solution for this is to remove the synchronization delay chain and subtract with 0 delay. This could work if assumed that DC offset is more or less stable.

FSK demodulator

FSK is basicaly like FM, but with clear deviationf for 1 and 0.

This chapter gives an example on how to build FSK demodulator with Pyha. Goal of this chapter is to show how previously built complex blocks can be connected togather in an easy way.

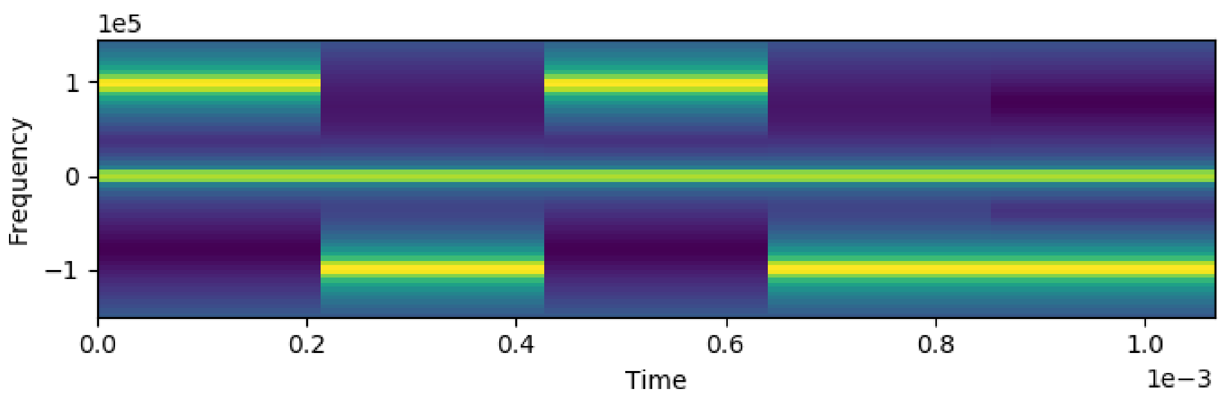

Fig. 6.10: Sample FSK spectrum, 1e5 deviation.

Fig. 6.10 show a spectrum of sample FSK spectrum. Carried data is `[1, 0, 1, 0, 0]`. As can be seen, for bit 1 there is positive frequency content ant fro bit 0 negative (relative to carrier).

In the process of demodulation, we would like to recover the bits from the frequency content. There are multiple ways to demodulate FM signal, for example Baseband Delay Demodulator (also known as quadrature demodulator) and using Phase-Locked loop *[6]*.

Most popular choice in the SDR world is the Quadrature Demodulaotr, since signal is already at baseband and it does not contain feedback loops. *[6]* shows that this demodulator has better performance comparet to PLL method.

Quadrature Demodulator involves some complex arithmetic like complex multiplyer and arcsin calculation. The purpos of this chapter is not to go into details but rather show how such kind of block could be used in Pyha.

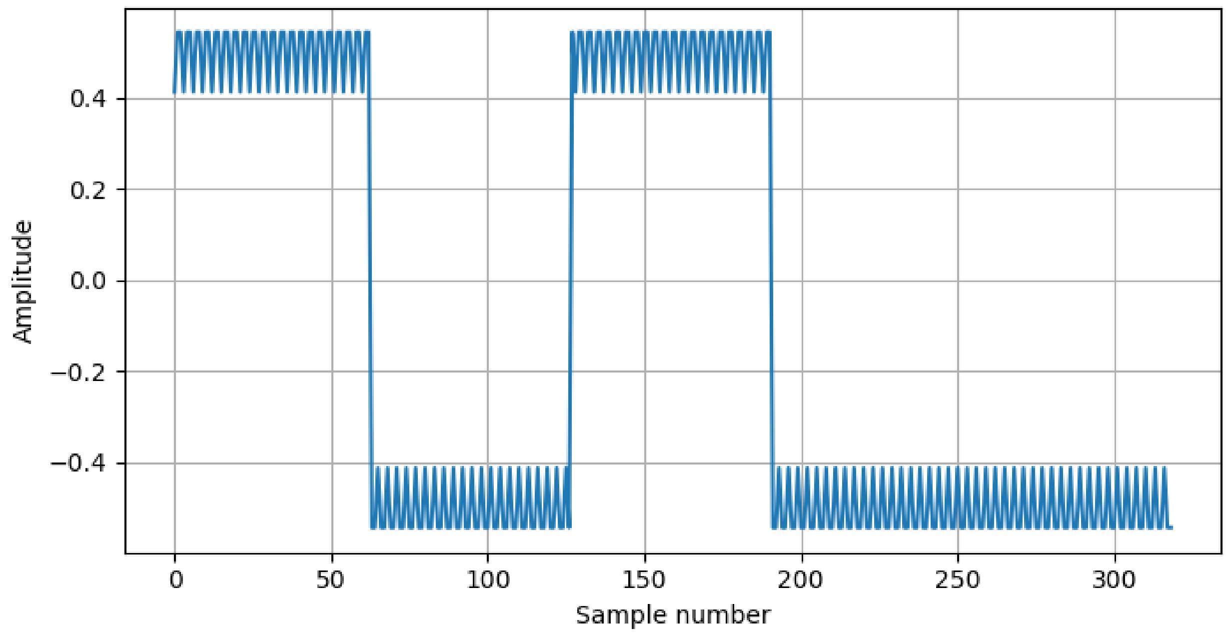

Fig. 6.11: Output of Qaudrature Demodulator

Fig. 6.11 show the Quadrature Demodulator output where input is the signal shown in Fig. 6.10. Note that the result looks already like digital signal. Result is a bit noisy as the input was noisy aswell.

Next step in the demodulator path is matched filtering. Since we are dealing with squared signals we can use the moving average algorithm for this purpose.

Implementation with Pyha

Implementation is rather straight forward, as shown on Fig. 6.7, algorithm must run input signal over multiple moving average filters (that we have already implemented in previous chapter) and then substract the filter chain output of the delayed input signal.

Listing 6.6: Parametrizable demodulator

```
class FSKDemodulator(HW):
    def __init__(self, deviation, fs, sps):
        self.demod = QuadratureDemodulator(self.gain)
        self.match = MovingAverage(sps)
        self._delay = self.demod._delay + self.match._delay
```

```
    def main(self, input):
        demod = self.demod.main(input)
        match = self.match.main(demod)
        return match

    def model_main(self, input_list):
        demod = self.demod.model_main(input_list)
        match = self.match.model_main(demod)
        return match
```

Listing 6.6 shows the Python implementation. Overall it is a pretty straigth forward Python code. Quadrature demodulator and Moving average are defined in the constructor bit, then 'main' and 'model main' make use of them.

One thing to note that the `model_main` and `main` are nearly identical. That shows that Pyha has archived one of the goals by simplifying hardware design portion.

Unit test for this module have not been listed as most of the testing is done in Ipython Notebook environment, as written in some chapter Pyha is capable or collecting these tests for unit-testing. Can be seen here.

Resource usage

RTL is too big to include a screenshot, project can be opened here..

Synhesizing with Quartus gave following resorce usage:

- Total logic elements: 1,499 / 39,600 (4 %)
- Total memory bits: 36 / 1,161,216 (< 1 %)
- Embedded multipliers: 10 / 232 (4 %)

Maximum reported clock speed is 173 MHz (standard compilation).

Conclusions

This chapter showed how to use existing Pyha components to synthesise complex system.

Further improvements

Next step would be to add some sort of clock recovery component in order to sample te bits.

Chapter 7

Conclusion

This work studied the feasability of implementing direct Python to VHDL converter. Result is a way of converting Python object-oriented code into VHDL. It was described how this conversion was made and what tradeoffs had to been taken.

In addition, fixed-point type was developed to support conversion of floating point models. Automatix conversion to fixed-point was discussed.

Experimental compiler also bests the simulation/testing/verification side of HW development. By providing simple functions that can run all simulations at once, this enables to use well known unit test platforms like PyTest.

Lastly we showed that Pyha is already usable to convert some mdeium complexity designs, like FSK demodulator, that was used on Phantom 2 stuff..

Limitations/future work

Long term goal is to implement more DSP blocks, especially by using GNURadio blocks as models. In future it may be possible to turn GNURadio flow-graphs into FPGA designs, assuming we have matching FPGA blocks available.

Currently designs are limited to one clock signal, decimators are possible by using Streaming interface. Future plans is to add support for multirate signal processing, this would involve automatic PLL configuration. I am thinking about integration with Qsys to handle all the nasty clocking stuff.

Synthesizability has been tested on Intel Quartus software and on Cyclone IV device (one on BladeRF and LimeSDR). I assume it will work on other Intel FPGAs as well, no guarantees.

Fixed point conversion must be done by hand, however Pyha can keep track of all class and local variables during the simulations, so automatic conversion is very much possible in the future.

Integration to bus structures is another item in the wish-list. Streaming blocks already exist in very basic form. Ideally AvalonMM like buses should be supported, with automatic HAL generation, that would allow design of reconfigurable FIR filters for example.

Bibliography

[1] Pong P. Chu. *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*. Wiley, 2006.

[2] Steven W. Smith. *The Scientist & Engineer's Guide to Digital Signal Processing*. California Technical Pub, 1997.

[3] Altera. Cyclone iv fpga device family overview. 2016.

[4] BladeRF community. Dc offset and iq imbalance correction. 2017. URL: https://github.com/Nuand/bladeRF/wiki/DC-offset-and-IQ-Imbalance-Correction.

[5] Rick Lyons. Linear-phase dc removal filter. 2008. URL: https://www.dsprelated.com/showarticle/58.php.

[6] Christoph Haller Franz Schnyder. Implementation of fm demodulator algorithms on a high performance digital signal processor. Master's thesis, Nanyang Technological University, 2002.

[7] David Bishop. Fixed point package user's guide. 2016.

[8] Python documentation. URL: https://docs.python.org.