Thesis

Gaspar Karm

Apr 04, 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

1.1 Objective/goal

Testing and verifying is hard. The goal of this study is to implement experimental Python to VHDL compiler. Provide an model and unit test based workflow, where tests that are defined for the model can be reused for RTL and GATE level simulations.

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.

1.2 Scope

??? Focus on LimeSDR board and GnuRadio Pothos, frameworks.

1.3 Structure

First chapter gives an short background about the context of this thesis and existing toolsets that provide conversion from higher level languages to Gates.

Chapter 2

Background

Give a short overview of whats up.

2.1 A structured VHDL design method

The base of this thesis builds on top of the work of Jiri Gaisler about 'Structured VHDL design method' *[1]*. This chapter gives an overview of what it is about.

2.1.1 Introduction

The VHDL language [22] was developed to allow modelling of digital hardware. It can be seen as a super-set of Ada, with a built-in message passing mechanism called sig- nals. When the language was first put to use, it was used for high-level behavioural simula- tion only. 'Synthesis' into VLSI devices was made by manually converting the models into schematics using gates and building blocks from a target library. However, manual conversion tended to be error-prone, and was likely to invalidate the effort of system simulation. To address this problem, VHDL synthesis tools that could convert VHDL code directly to a technology netlist started to emerge on the market in the begining of 1990's. Since the VHDL code could now be directly synthesised, the development of the models was primarily made by digital hardware designers rather than software engi- neers. The hardware engineers were used to schematic entry as design method, and their usage of VHDL resembled the dataflow design style of schematics. The functionality was coded using a mix of concurrent statments and short processes, each decribing a limited piece of functionality such as a register, multiplexer, adder or state machine. In the early 1990's, such a design style was acceptable since the complexity of the circuits was relatively low ($<$ 50 Kgates) and the synthesis tools could not handle more complex VHDL structures. However, today the device complexity can reach several millions of gates, and the synthesis tools accept a much larger part of the VHDL standard. It should therefore be possible to use a more modern and efficient VHDL design

method than the traditional 'dataflow' version. This chapter will describe such a method and compare it to the 'dataflow' version. *[1]*

2.1.2 The problems with the 'dataflow' design method

The most commonly used design 'style' for synthesisable VHDL models is what can be called the 'dataflow' style. A larger number of concurrent VHDL statements and small processes connected through signals are used to implemenet the desired function- ality. Reading and understanding dataflow VHDL code is difficult since the concurrent statements and processes do not execute in the order they are written, but when any of their input signals change value. It is not uncommon that to extract the functionality of dataflow code, a block diagram has to be drawn to indentify the dataflow and depend- ecies between the statements. The readability of dataflow VHDL code can compared to an ordinary schematic where the wires connecting the various blocks have been removed, and the block inputs and outputs are just labeled with signal names! *[1]*

A problem with the dataflow method is also the low abstraction level. The functionality is coded with simple constructs typically consisting of multiplexers, bit-wise operators and conditional assignments (if-then-else). The overall algorithm might be very difficult to recognize and debug. *[1]*

2.1.3 The goals and means of the 'two-process' design method

To overcome the limitations of the dataflow design style, a new 'two-process' coding method is proposed. The method is applicable to any synchrounous single-clock design, which represents the majority of all designs. The goal of the two-process method is to:

- Provide uniform algorithm encoding
- Increase abstraction level
- Improve readability
- Clearly identify sequential logic
- Simplify debugging
- Improve simulation speed
- Provide one model for both synthesis and simulation

The above goals are reached with suprisingly simple means:

- Using record types in all port and signal declarations
- Only using two processes per entity
- Using high-level sequential statements to code the algorithm

The following section will outline how the two-process method works and how it com- pares with the traditional dataflow method. *[1]*

2.1.4 Using two processes per entity

The biggest difference between a program in VHDL and standard programming lan- guage such C, is that VHDL allows concurrent statements and processes that are sched- uled for execution by events rather than in then order they are written. This reflects indeed the dataflow behaviour of real hardware, but becomes difficult to understand and analyse when the number of concurrent statments passes some threashold (e.g. 50). On the other hand, analysing the behaviour of programs witten in sequential program- ming languages does not become a problem even if the program tends to grow, since there is only one thread of control and execution is done sequentially from top to bot- tom. In order to improve readability and provide a uniform way of encode the algorithm of a VHDL entity, the two-process method only uses two processes per entity: one process that contains all combinational (asynchronous) logic, and one process that contains all sequential logic (registers). Using this structure, the complete algorithm can be coded in sequential (non-concurrent) statements in the combinational process while the sequential process only contains registers, i.e. the state.:cite:*structvhdl_gaisler*

2.1.5 Other improvements

Gaisler also shows how to use records to group all the registers into one variable and use records for port connections to improve design hirarcy. In addtion, Gaisler shows that higher level constructs like sub-programs and loop statements are fully usable and syhtnesisable.

Comparison MEC/LEON: *[1]*

ERC32 memory contoller MEC

- Ad-hoc method (15 designers)
- 25,000 lines of code
- 45 entities, 800 processes
- 2000 signals
- 3000 signal assigments
- 30 Kgates, 10 man-years,numerous of bugs, 3 iterations

LEON SPARC V8 processor

- Two process method (mostly)
- 15,000 lines of code
- 37 entities, 75 processes

- 300 signals
- 800 signal assignments
- 100k gates, 2 man-years,
- no bugs in first silicon

2.1.6 Summary and conclusions

The presented two-process method is a way of producing structured and readable VHDL code, suitable for efficient simulation and synthesis. By defining a common coding style, the algorithm can be easily identified and the code analysed and main- tained also by other engineers than the main designer. Using sequential VHDL state- ments to code the algorithm also allows the use of complex statements and a higher abtraction level. Debugging and analysis is simplified due to the serial execution of statements, rather than the parallel flow used in dataflow coding.:cite:*structvhdl_gaisler*

2.2 Contributions of this work

First part of this work builds on top of the Jiri Gaisler work, but makes significant improvements. First this work adds synthesisable object-orientational support to VHDL language.

Next we provide a way of signal assignment that can be written without the use of VHDL signal assignment semantics. The point of removing this is to make the programming model more structured, and standard.

Lastly this work provides an Python to VHDL mapping, in order to speed up development time also the Python program can be simulated.

2.3 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

2.4 HDL related tools in Python

As the idea of converting higehr level languages to VHDL/Verilog is not new, this chapter gives an overview of previous works and states how current work differs from them.

2.4.1 MyHDL

MyHDL is Python to VHDL/Verilog converter, first release dating back to 2003. It turns Python into a hardware description and verification language, providing hardware engineers with the power of the Python ecosystem.:cite:*myhdlweb*

MyHDL has been used in the design of multiple ASICs and numerious FPGA projects.:cite:*myhdlfelton*

MyHDL, like VHDL and Verilog,is a hardware description language. MyHDL does not include "IP" or cores directly *[2]*.

MyHDL is not a tool to take arbitrary Python code and create working hardware [7]. MyHDL is similar to existing HDLs; the convertible subset of the language describes hardware behavior at the Register Transfer Level (RTL) of abstraction. Clearly, this indicates MyHDL is not a HighLevel Synthesis (HLS) language. *[2]*

MyHDL works with data-flow paradigm, not good, not good.

Example

Here is a simple example of describing and register with MyHDL. Listing 2.1 shows an register code in MyHDL. One thing to note is that it uses Python function as a base unit and `always` blocks, that all is very similiar to Verilog language, clearly this infers a process with separate clock and reset signals.

Another thing to note is the assignment of 'q' value. It uses the 'next' value. Pyha steals this.

Listing 2.1: Register in MyHDL *[3]*

```
from myhdl import *

def dffa(q, d, clk, rst):

    @always(clk.posedge, rst.negedge)
    def logic():
        if rst == 0:
            q.next = 0
        else:
            q.next = d
```

```
    return logic
```

Listing 2.2 shows the code required to simulate the design. It is not important to understand what goes on, but to see that simulating in MyHDL is not simple. It requires the user to handle clock and reset etc. Dataflow principles even in testbench.

Listing 2.2: Register in MyHDL *[3]*

```
from random import randrange

def test_dffa():

    q, d, clk, rst = [Signal(bool(0)) for i in range(4)]

    dffa_inst = dffa(q, d, clk, rst)

    @always(delay(10))
    def clkgen():
        clk.next = not clk

    @always(clk.negedge)
    def stimulus():
        d.next = randrange(2)

    @instance
    def rstgen():
        yield delay(5)
        rst.next = 1
        while True:
            yield delay(randrange(500, 1000))
            rst.next = 0
            yield delay(randrange(80, 140))
            rst.next = 1

    return dffa_inst, clkgen, stimulus, rstgen

def simulate(timesteps):
    tb = traceSignals(test_dffa)
    sim = Simulation(tb)
    sim.run(timesteps)

simulate(20000)
```

Problems with MyHDL

- Writing testbenches is hard, dataflow is bad, have to handle clock and reset
- Conversion very limited (jan rant)

Convertable subset is extreamly limited compared to the simulatable subset. Many users (including me) have been dissapointed about this, this has even led the author of MyHDL, Jan Decaluwe to write an blog post about how MyHDL is 'simulation-oriented language' *[4]*.

2.4.2 Migen

Migen is a Python-based tool that aims at automating further the VLSI design process. Migen makes it possible to apply modern software concepts such as object-oriented programming and metaprogramming to design hardware. This results in more elegant and easily maintained designs and reduces the incidence of human errors. *[5]*

Despite being faster than schematics entry, hardware design with Verilog and VHDL remains tedious and inefficient for several reasons. The event-driven model introduces issues and manual coding that are unnecessary for synchronous circuits, which represent the lion's share of today's logic designs. Counter- intuitive arithmetic rules result in steeper learning curves and provide a fertile ground for subtle bugs in designs. Finally, support for procedural generation of logic (metaprogramming) through "generate" statements is very limited and restricts the ways code can be made generic, reused and organized. *[5]*

To address those issues, we have developed the Migen FHDL library that replaces the e vent-driven paradigm with the notions of combinatorial and synchronous statements, has arithmetic rules that make integers always behave like mathematical integers, and most importantly allows the design's logic to be constructed by a Python program. This last point enables hardware designers to take advantage of the richness of the Python language - object oriented programming, function parameters, generators, operator overloading, libraries, etc. - to build well organized, reusable and elegant designs. *[5]*

Other Migen libraries are built on FHDL and provide various tools such as a system-on-chip interconnect infrastructure, a dataflow programming system, a more traditional high-level synthesizer that compiles Python routines into state machines with datapaths, and a simulator that allows test benches to be written in Python. *[5]*

- Python as a meta-language for HDL
- Restricted to locally synchronous circuits (multiple clock domains are supported)
- **Designs are split into:**
  - synchronous statements
  - combinatorial statements

- Statements expressed using nested Python objects

[6]

Has some advanced features like BUS support:

- Wishbone1
- SRAM-like CSR
- DFI 2
- LASMI

[6]

Able to generate hardware abstraction layer in C, for bus usage

The base idea is very similiar to of Pyha, to get rid of dataflow/event driven modeling. It has a very strange way of programming. Pyha has clear edge here. Simulation in Python support..looks weak, it relies more on Verilog simulator

Many systems build with this system. Now has more github stars then MyHDL.

Example

Listing 2.3 showns a LED blinker module implemented in Migen, it consists of a counter that when finished toggles the LED state.

As written before, Migen separates hardware design into combinatory and synch parts. What can be seen is kind of a metaprogramming. That is in migen one cannot write `counter = period` but have to write `counter.eq(period)`, same goes for if statements etc. That is the price you have to pay in order to use Migen.

Much bigger problem of this approach is that the hardware part of the code is basically not debuggable. Migen supports some kind of Python simulator but it is not much better than MyHDL one.

Listing 2.3: Register in MyHDL *[5]*

```
class Blinker(Module):
    def __init__(self, led, maxperiod):
        counter = Signal(max=maxperiod+1)
        period = Signal(max=maxperiod+1)
        self.comb += period.eq(maxperiod)
        self.sync += If(counter == 0,
                led.eq(~led),
                counter.eq(period)
            ).Else(
                counter.eq(counter - 1)
            )
```

Problems with MiGen

Migen is awesome but it also has some problems.

- Simulation is not easy,
- Not debuggable in Python domain

2.4.3 Cocotb

EDAPLAYGROUND Cocotb is a COroutine based COsimulation TestBench environment for verifying VHDL/Verilog RTL using Python. *[7]*

Unlike MyHDL and Migen, Cocotb is not a Python to HDL converter, instead it is meant to simulate VHDL/Verilog designs.

A typical cocotb testbench requires no additional RTL code. The Design Under Test (DUT) is instantiated as the toplevel in the simulator without any wrapper code. Cocotb drives stimulus onto the inputs to the DUT (or further down the hierarchy) and monitors the outputs directly from Python. *[7]*

A test is simply a Python function. At any given time either the simulator is advancing time or the Python code is executing. The yield keyword is used to indicate when to pass control of execution back to the simulator. A test can spawn multiple coroutines, allowing for independent flows of execution.

Problems with Cocotb

Major problem with Cocotb is that the tests are to be written to test the HDL part only. Often it happens that there is also some higher level model that could use unit-testing. With Cocotb one would need to develope two sets of tests, one for the model and another for HDL, this situation is bound to end badly.. unsynchronzed model and HDL.

Minor headace is that Cocotb runs Python test file started from C program, meaning that for debugging one has to use remote debugger, that is not very convenient.

2.5 Other HDLS

This thesis focuses on the Python to VHDL conversion. There exsist however many more tools that instead of Python convert something else to VHDL/Verilog.

In this paper we introduce Chisel , a new hardware construction language that supports advanced hardware design using highly parameterized generators and layered domain-specific hardware languages. By embedding Chisel in the Scala programming language, we raise the level of hardware design ab straction by providing concepts including object orientation,

functional programming, parameterized types, and type inference. Chisel can generate a high-speed C++-based cycle-accurate software simulator, or low-level Verilog designed to map to either FPGAs or to a standard ASIC flow for synthesis. This paper presents Chisel, its embedding in Scala, hardware examples,

> and results for C++ simulation, Verilog emulation and ASIC synthesis. *[8]*

Now there is a new version called Chisel3, that seemingly has not gained much ground yet.

Also there is a spinoff project called SpinalHDL that tries to fix many shortcomings of Chisel. No sim support?

These are converters written in Scala. They seem to be very feature rich. Chisel developed by University of California. Big acceptance on writing RISC instruction set processors.

CλaSH (pronounced 'clash') is a functional hardware description language that borrows both its syntax and semantics from the functional programming language Haskell. It provides a familiar structural design approach to both combinational and synchronous sequential circuits. The CλaSH compiler transforms these high-level descriptions to low-level synthesizable VHDL, Verilog, or SystemVerilog.

Features of CλaSH:

- Strongly typed, but with a very high degree of type inference, enabling both safe and fast prototyping using concise descriptions.
- Interactive REPL: load your designs in an interpreter and easily test all your component without needing to setup a test bench.
- Compile your designs for fast simulation.
- Higher-order functions, in combination with type inference, result in designs that are fully parametric by default.
- Synchronous sequential circuit design based on streams of values, called Signals, lead to natural descriptions of feedback loops.
- Multiple clock domains, with type safe clock domain crossing.
- Template language for introducing new VHDL/(System)Verilog primitives.

[9]

https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_1/ug902-vivado-high-level-synthesis.pdf

Todo. maybe skip?

Chapter 3

VHDL extensions

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.

Lay down a common ground on which VHDL and Python coold be connected.

While other HDL converters use VHDL/Verilog as low level conversion target. Pyha goes other way around, as shown by the Gardner study:cite:*structvhdl_gaisler*, VHDL language can be used with quite high level progrmaming constructs. Pyha tries to take advantage of this.

This chapter tries to enchance the VHDL language with some basic Python elements in order to provide some common ground for the conversion task.

Disadvantage is that it can be only converted to VHDL. Advantages are numerous:

- Similiar code in VHDL and Python
- Clean conversion output
- Easy to use VHDL Fixed point package

3.1 Object-oriented model in VHDL

As stated by the goal of this work, converting Object-oriented designs into HDL. While it may seem that VHDL has no support for OOP, it is actually not true.

There have been previous study regarding OOP in VHDL before. In *[17]* proposal was made to extend VHDL language with OOP semantics, this effort ended with development of OO-VHDL *[18]*, that is VHDL preprocessor that could turn proposend extensions to standard VHDL. This work was done in ˜2000, current status is unknown, it certanly did not make it to the VHDL standard.

While the *[18]* tried to extend VHDLs data-flow side of OOP, there actually exsists another way to do it, that is inherited from ADA.

VHDL supports 'packages' to group common types and functions into one namespace. Package in VHDL must contain an declaration and body (this is the same concept as header and source files in C).

Listing 3.1: OOP in VHDL

```
package ExamplePackage is

    type self_t is record
        var: integer;
    end record;

    procedure set_var(self:inout self_t; new_var: integer);
    procedure get_var(self:inout self_t; ret_0:out integer);
end package;

package body ExamplePackage is

    procedure set_var(self:inout self_t; new_var: integer) is
    begin
        self.var := new_var;
    end procedure;

    procedure get_var(self:inout self_t; ret_0:out integer) is
    begin
        ret_0 := self.var
    end procedure;

end package body;
```

Note: VHDL also supports 'functions' that can return a value, but these are not suitable for using with class model as they have no 'inout' parameter to handle the object datamodel.

Listing 3.1 gives basic example on how to write OOP in VHDL. Base point of OOP is to define some data and then functions that can perform operations with this data structure. In the example we have used 'record' (like struct in C) to construct an datamodel for the object, to keep it simple it only consists of one integer variable.

In addition, simple setter function is provided, that takes as a first parameter the datamodel object and sets the integer variable to the second argument. It also provides a getter function, VHDL procedures cannot `return` values, but can use `out` arguments as outputs, this is convenient as it allows returning multiple values.

This method of writing OOP code is quite common in C also, principle is the same. Make a structure to hold the datamodel and then always pass this structure as the first parameter to functions.

3.2 Synthesising combinatory logic

A combinational circuit, by definition, is a circuit whose output, after the initial transient period, is a function of current input. It has no internal state and therefore is "memoryless" about the past events (or past inputs) *[10]*. In other words, combinatory circuits have no registers, i like to call it 'stuff between registers'.

OOP-VHDL shown on Listing 3.1 will probably look useless to anyone who has VHDL experience. First reaction is probably that this thing is not synthesizeable.

Here we show that this simple example is already good enough to synthesize combinatory logic.

Todo

Example of synthesisying some combinatory stuff

One thing to note is that the object side of this example is quite useless, we can use it only to store constants.

Actually sequential logic could be inferred by guaranteeing that the class object values are always read before written into. But this is an extreamly error prone way of inferring registers. *[10]*

3.3 Working with registers

A sequential circuit, on the other hand, has an internal state, or memory. Its output is a function of current input as well as the internal state. The internal state essentially "memorizes" the effect of the past input values. The output thus is affected by current input value as well as past input values (or the entire sequence of input values). That is why we call a circuit with internal state a sequential circuit. *[10]*

Todo

dff image?

Point here is that the design contains registers, these are memory elements that are controlled by the clock signal.

Register has one input and one output. It outputs the current value stored in the memory. Input is used to take the next value. Note that the input is only sampled on the clock edge.

VHDL has a special assigment to work with such kind of constructs, it is signal assignment. Basically signal assigmnet is

Listing 3.2: VHDL signal assignment

```
a <= b;
c <= a;
```

Listing 3.2 shows VHDL signal assignment in action. First value of 'b' is assigned to 'a' and then 'a' assigned to 'c'. Now the problem with these assignments are that they work in a weird way, namely a is not actually assigned b, and c is not assigned a. bla bla bla.

Listing 3.3: Better VHDL signal assignment

```
a.next := b;
c.next := a;
```

Listing 3.3 shows a more clear way of what is going on. Note that this uses regular assignment operator. Assuming 'a' and 'c' are objects that have next variable.

Using 'next' attribute for signal assignment is now used in literally every other HDL than Verilog/VHDL

Author of MyHDL package has written a good writeup on how it handles signal assigment [jan_myhdl_signals], in short they use the same 'next' idiom. Even Pong P. Chu, author of one of the best VHDL books, teaches the reader to write registers with two variables, one for the current value and another one for 'next'.

Using an signal assigment inside a clocked process always infers a register.

3.4 Getting rid of signal assignments

We would like to save registers as our class object values, and to get rid of signal assignment.

Much better way to work with registers is to embrace the style popularized by MyHDL, that is signal is an object that has a current value and 'next' value.

3.5 Synthesisability

3.6 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.

3.7 Conclusions

This chapter shows how to OOP in VHDL, we demonstrate that the approach is fully synthesisable.

Chapter 4

Conversion to Python

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

What about verilog?

While other high-level tools decide to use VHDL/Verilog as low level conversion target. Pyha goes other way around, as shown by the Gardner study, VHDL language can be used with quite high level progrmaming constructs. Pyha tries to take advantage of this. Disadvantage is that it can be only converted to VHDL. Advantages are numerous:

- Similiar code in VHDL and Python
- Clean conversion output
- ?

4.1 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.

4.2 Comparison of syntax

4.3 Assignments

4.3.1 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. *[10]*

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: *[10]*

4.3.2 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.

4.4 Design resuse

4.5 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
```

`this-py` 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

```
1  type self_t is record
2      instance_member: integer;
3  end record;
4
5  procedure main(self:inout self_t; a: integer; ret_0:out integer) is
6  begin
7      self.instance_member := a;
8      ret_0 := self.instance_member;
9      return;
10 end procedure;
```

4.6 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;
```

4.6.1 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);
```

4.7 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

4.8 Conclusions

This chapter showed how Python OOP code can be converted into VHDL OOP code.

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.

5.1 Convetional design flow

VHDL uuendused? VUNIT VUEM?

5.2 Test-driven development / unit-tests

5.3 Model based development

How MyHDl and other stuffs contribute here?

5.4 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.

5.4.1 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.

5.4.2 Ipython notebook

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

5.5 Conclusion

It is clear that Pyha provides many conveneince functions to greatly simplyfy the testing of model based designs.

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.

6.1 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. *[11]*

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.

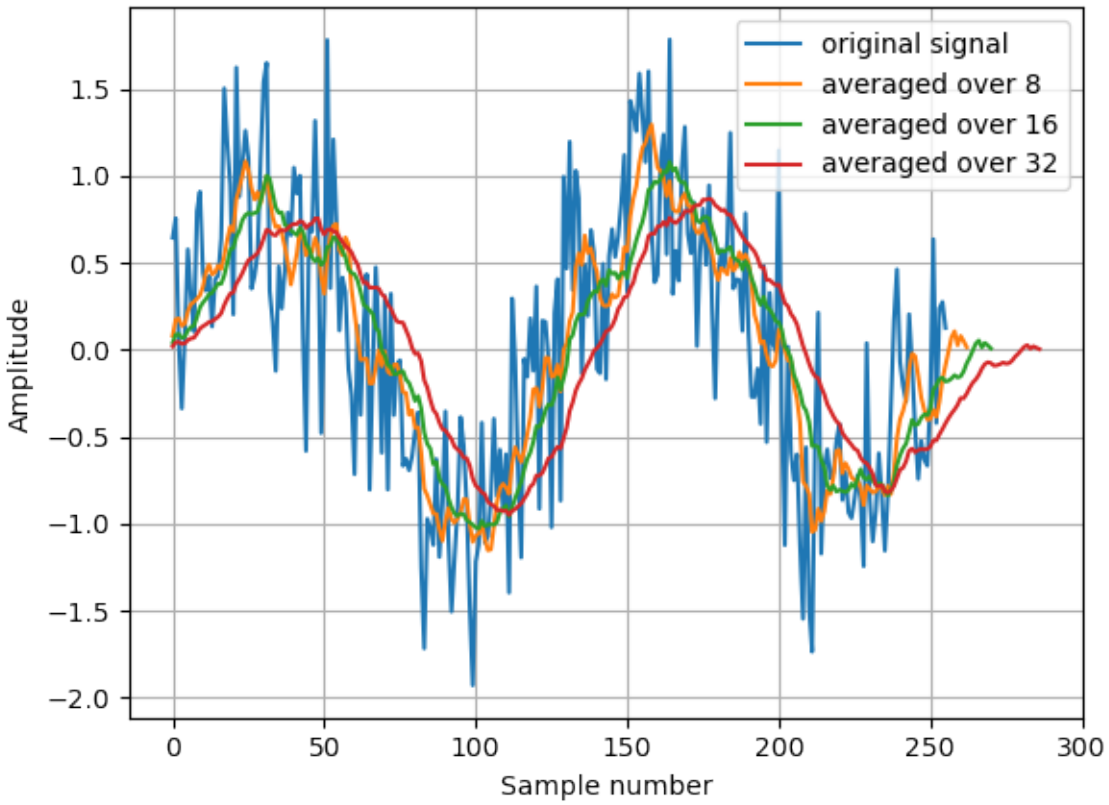

Fig. 6.1: Example of moving averager as noise reduction

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 *[11]*. 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).

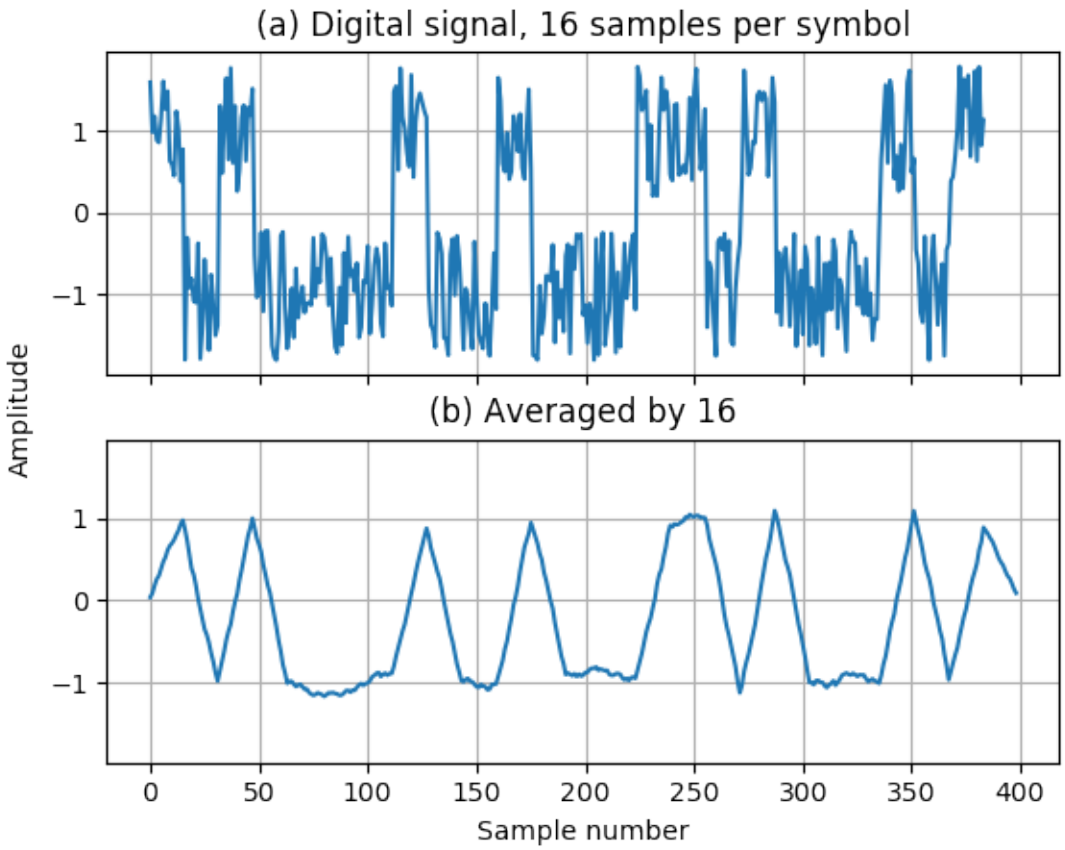

Fig. 6.2: Moving average as matched filter

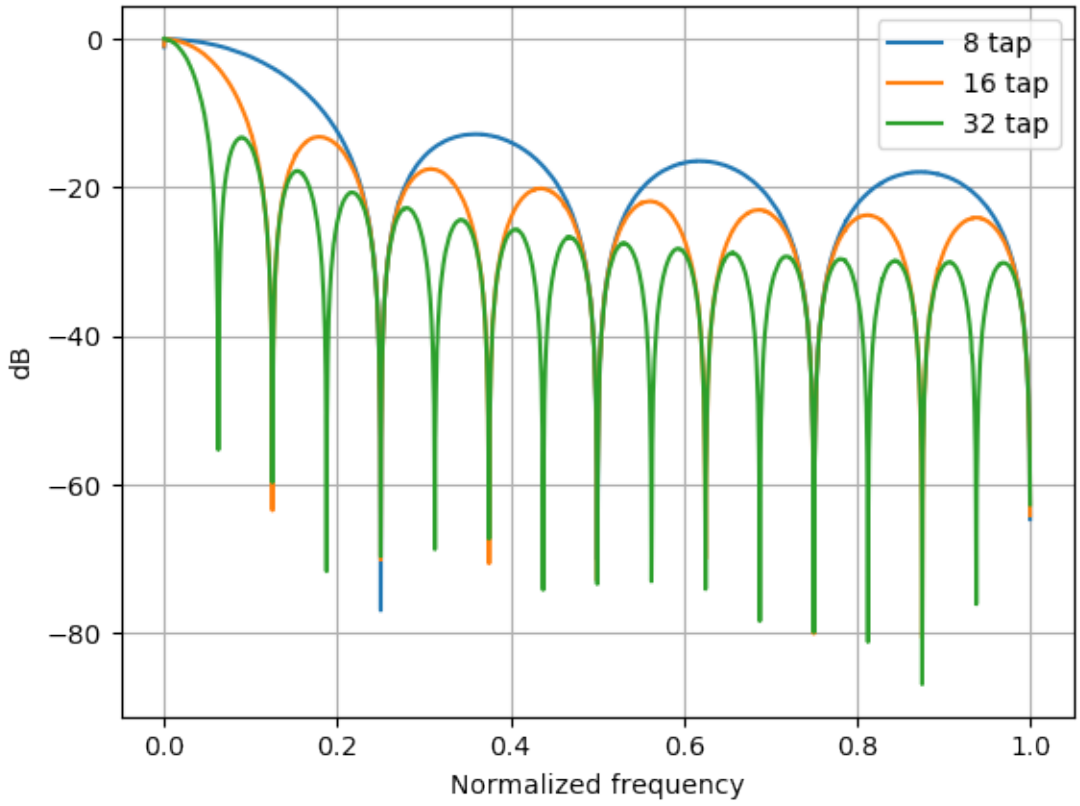

Fig. 6.3: Frequency response of moving average filter

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). *[11]*

6.1.1 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.

6.1.2 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:

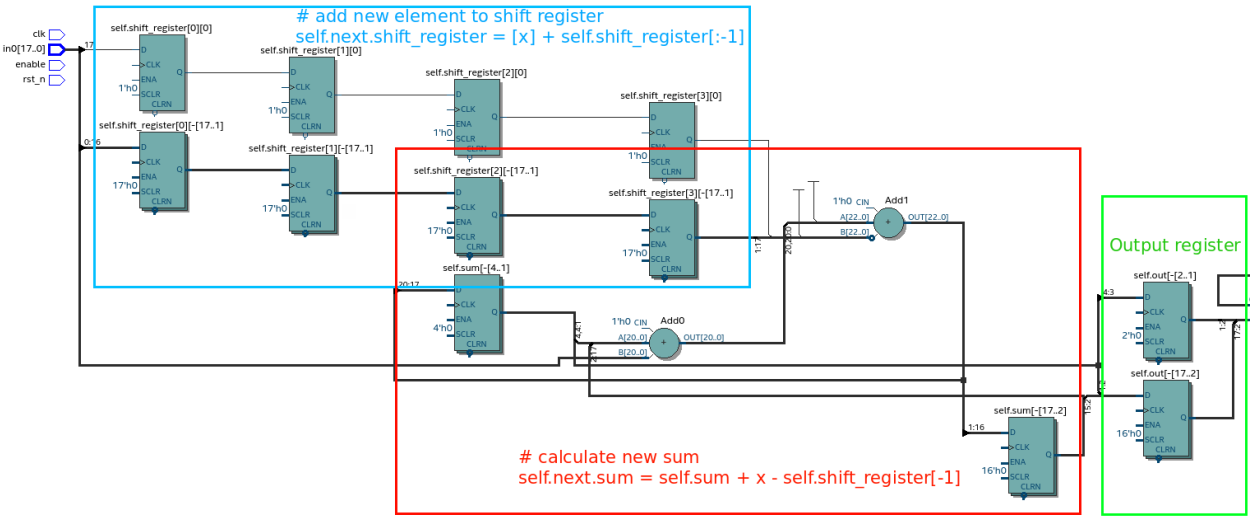

Fig. 6.4: RTL view of moving average (Intel Quartus RTL viewer)

Fig. 6.4 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 *[12]*:

[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 *[12]*.

6.2 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. *[13]*

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 *[14]* 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.

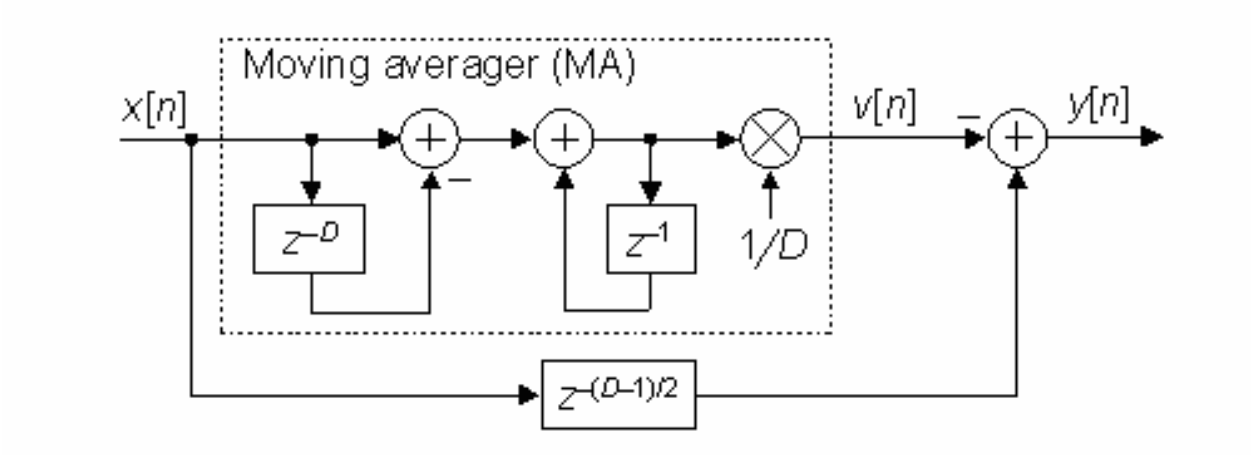

Fig. 6.5: Basic DC removal using moving averager *[14]*

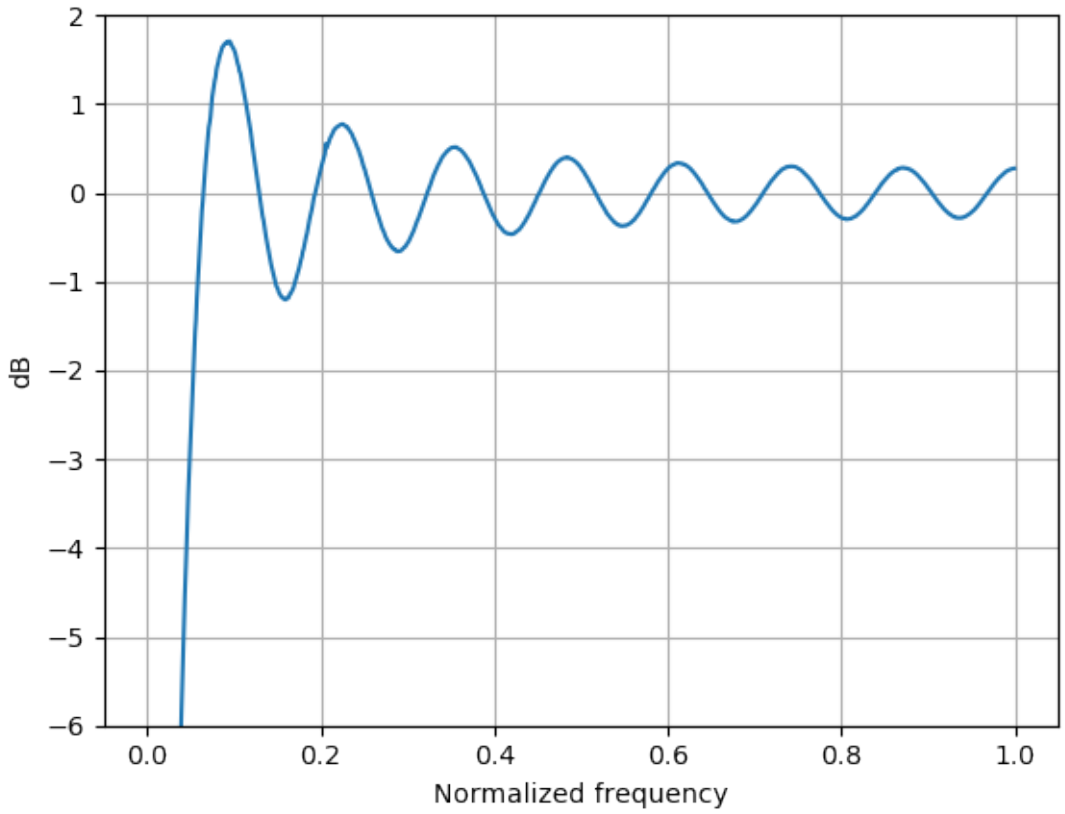

Fig. 6.6: Frequency response of DC removal circuit with Moving average length 31

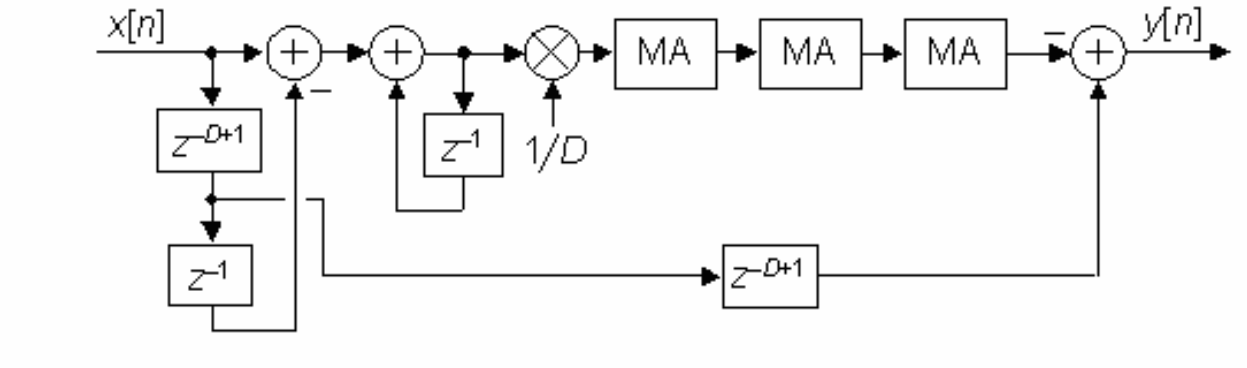

Fig. 6.7: Removing DC with chained moving averagers *[14]*

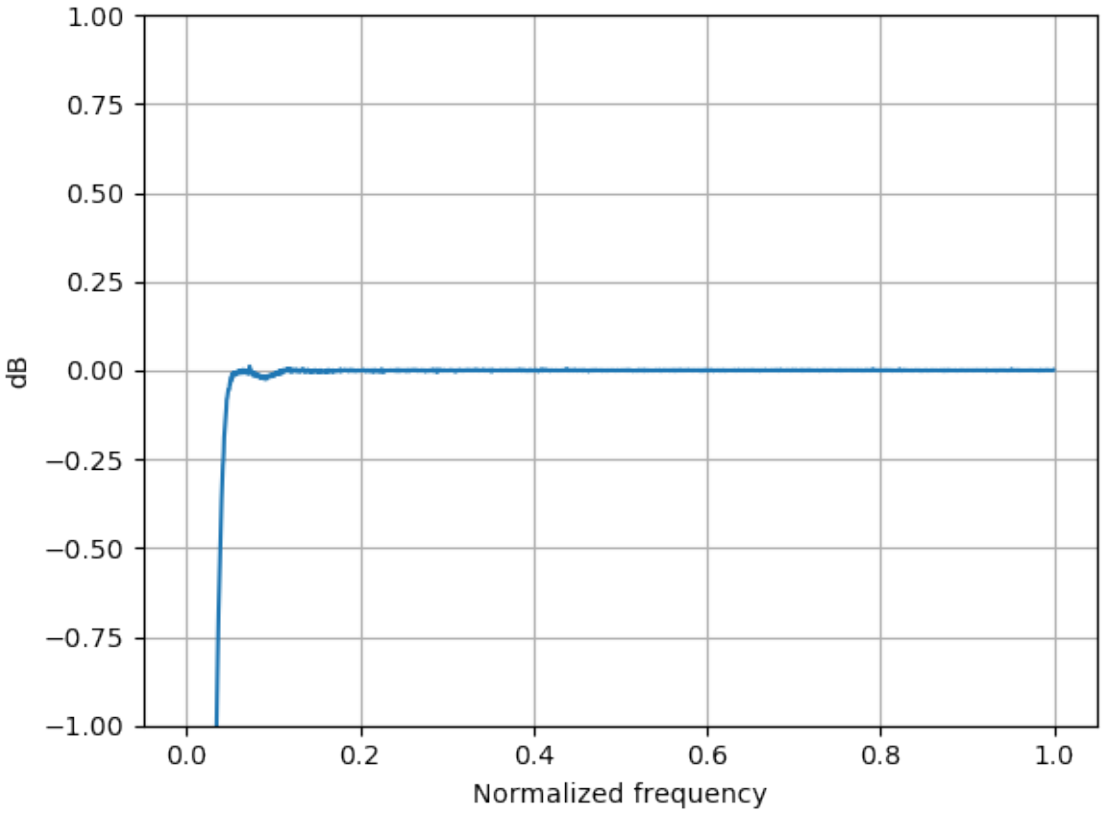

Fig. 6.8: Frequency response of DC removal, 4 cascaded moving averagers

6.2.1 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.

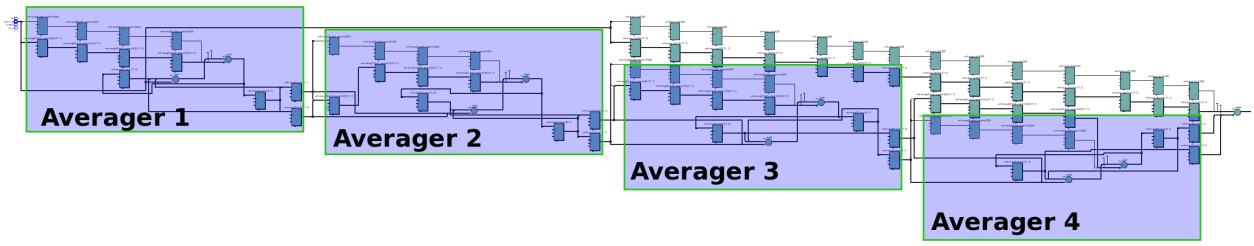

Fig. 6.9: RTL view of simplified DC-Removal (Intel Quartus RTL viewer)

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

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

6.2.2 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.

6.3 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.

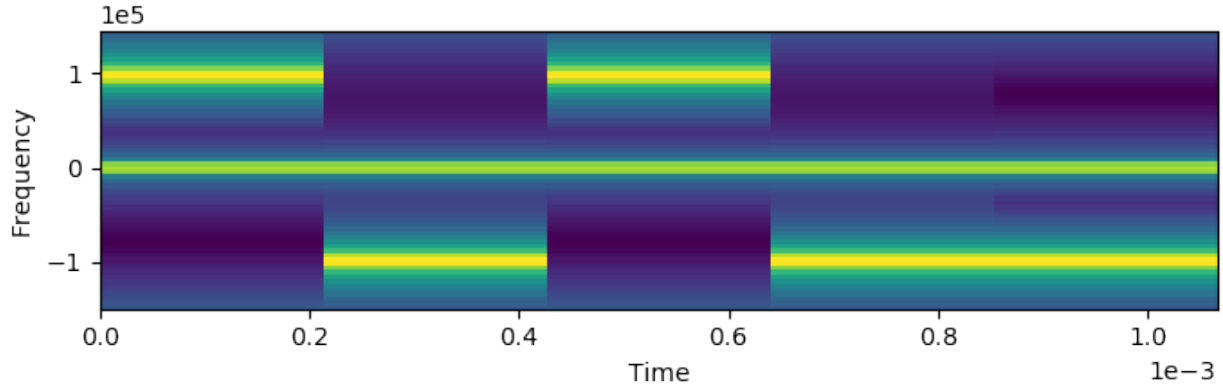

Fig. 6.10: Sample FSK spectrum, 1e5 deviation.

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 *[15]*.

Most popular choice in the SDR world is the Quadrature Demodulaotr, since signal is already at baseband and it does not contain feedback loops. *[15]* 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.

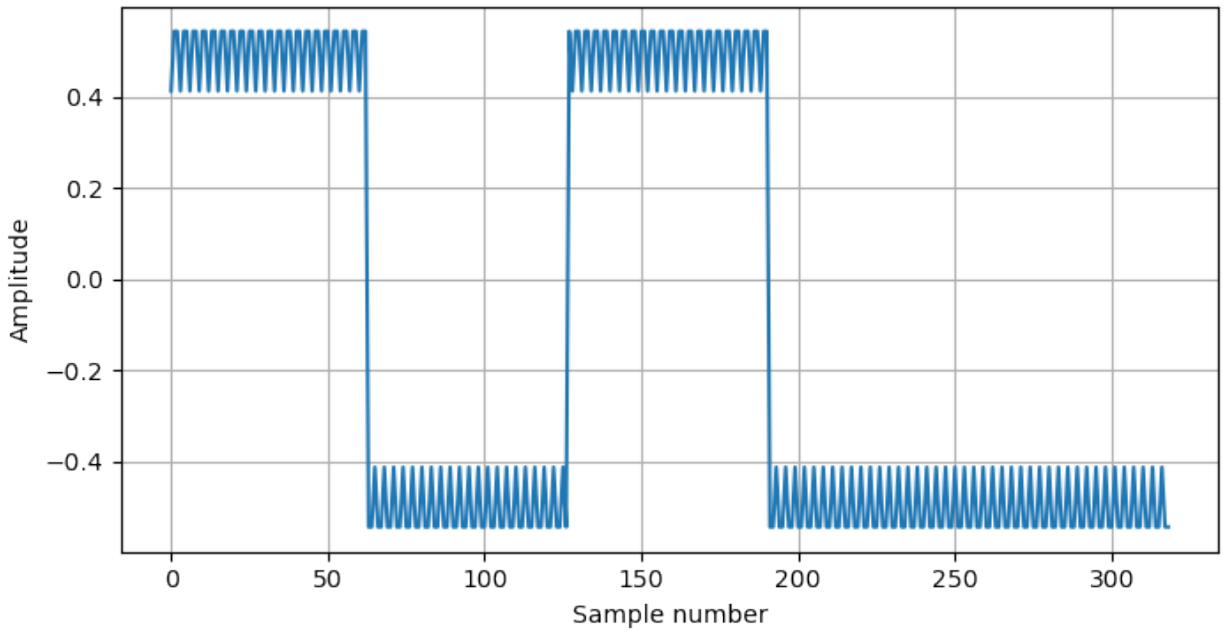

Fig. 6.11: Output of Qaudrature Demodulator

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.

6.3.1 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).

6.3.2 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..

7.1 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] Jiri Gaisler. A structured vhdl design method. URL: http://www.gaisler.com/doc/vhdl2proc.pdf.

[2] Christopher Felton. Why yoo should be using python/myhdl as your hdl. 2013.

[3] Myhdl. URL: http://www.myhdl.org.

[4] Jan Decaluwe. It's a simulation language! URL: http://www.jandecaluwe.com/blog/its-a-simulation-language.html.

[5] Migen. URL: https://m-labs.hk/gateware.html.

[6] Sebastien Bourdeauducq. Migen presentation. URL: https://m-labs.hk/migen/slides.pdf.

[7] PotentialVentures. Cocotb documentation. URL: cocotb.readthedocs.io.

[8] Jonathan Bachrach. Chisel: constructing hardware in a scala embedded language. 2012.

[9] Clash. URL: http://www.clash-lang.org/.

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

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

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

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

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

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

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

[17] Judith Benzakki and Bachir Djafri. *Object Oriented Extensions to VHDL, The LaMI proposal*, pages 334–347. Springer US, Boston, MA, 1997. URL: http://dx.doi.org/10.1007/978-0-387-35064-6_27, doi:10.1007/978-0-387-35064-6_27.

[18] S. Swamy, A. Molin, and B. Covnot. Oo-vhdl. object-oriented extensions to vhdl. *Computer*, 28(10):18–26, Oct 1995. doi:10.1109/2.467587.