Tallinn University of Technology
School of Information Technologies
Thomas Johann Seebeck Department of Electronics

Gaspar Karm

Pyha

Master's Thesis

Supervisors:
Muhammad Mahtab Alam
PhD

Yannick Le Moullec
PhD

Tallinn 2017

Contents:

Chapter 1

Introduction

Pyha is an new Python based hardware description language focusing on simplifying development and testing process of DSP systems. Pyha is an sequential HDL based on Python language. One part of the Pyha is the OOP VHDL model that helps converting simply to hardware. In a sense, Pyha can be considered as Python bindings for the object-oriented VHDL style, developed in this thesis.

Main features:

- Describe, test and simulate hardware in Python.
- Integration to run RTL and GATE simulations.
- Structured, sequential and object oriented designs
- Fixed point support and semi-automatic conversion from floating point
- Decent quality VHDL conversion output (get what you write, keeps hierarchy)
- Integration to Intel Quartus (run GATE level simulations)

1.1 Background

High-level synthesis (HLS) improves design productivity by automating the refinement from the algorithmic level to RTL *[8]*. In general the HLS world is dominated by C type languages, they are popular in both in commercially and academically, for example LegUp *[9]* is C based tool being developed at the University of Toronto.

Lately the Vivado HLS, developed by Xilinx, has been gaining some traction. As of 2015, it is included in the free design suite of Vivado, however that is device limited. Problem with HLS tools is that they often need manual code transformations and guidlines for the compiler in order to archive reasonable performance *[10]*.

On the other front there are projects that aim to raise the abstraction by improving the hardware description lannguages (HDL), working on the RTL level. For example, 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 *[11]*.

Chisel is an open-source hardware construction language developed at UC Berkeley that uses Scala programming language, they raise the level of hardware design abstraction by providing concepts including object orientation, functional programming, parameterized types, and type inference *[12]*.

DSP systems can be described in existing HLS and HDL languages, but currently the most popular ways is to use MATLAB/Simuling/HDLConverter flow. Simulink based design flow has been reported to be used in Berkeley Wireless Research Center (BWRC) *[13]*. Using this design flow, users describe their designs in Simulink using blocks provided by Xilinx System Generator *[13]*.

Traditional HDL languages like VHDL and Verilog are not standing still either. SystemVerilog (SV) is the new standard for Verilog language, it adds significant amount of new features to the language *[14]*. Most of the added synthesizable features already existed in VHDL, making the synthesizable subset of these two languages almost equal. In the design verification role, SystemVerilog(SV) is widely used in the chip-design industry. The big EDA (Cadence, Mentor, Synopsys) are pushing SV with Open Verification Methodology (OVM). At 2003, Aart de Geus, Synopsys CEO, has stated that SystemVerilog will replace VHDL `vhdl_dead`.

Meanwhile the VHDL development is going on mostly in the open-source sphere. Currently there is an VHDL-2017 standard in the works *[15]*. There are active work going on GHDL, that is open-source VHDL simulator. In addtion, lately more tools have been released like Open Source VHDL Verification Methodology (OSVVM) *[16]* and VUnit *[17]*, that simplifies unit-testing in VHDL.

1.2 Objective and scope

This thesis aims itself on the DSP systems. As mentioned in previos section, this domain is dominated by the MATLAB products. The tool, developed in the process of this thesis project, aims to provide an open-source alternative to the mostly MATLAB based DSP to hw flows.

The problem with MATLAB based workflows is the toolset cost. MATLAB has cleverly divided their products into multiple separade programs that each cost money. For HDL flow one would need MATLAB, Simulink, HDLCoder, HDLVerifier, DSPToolbox ... etc. The total price can easly go over 20000 EUR. In addition, the DSP flows require the use of FPGA Vendor synthesis tools and DSP generators like Xilinx System Generator, that furthermore cost 5000 EUR annually. Even if such kind of tooling could be afforded, the

designs are not shareable. Thus this way to completely unaceptable for open-source designs.

Thus the open-source designers must turn to alternative methods, for example in *[18]*, open-sourced a ADS-B decoder is implemented in hardware. In this work the authors first implement the model in MATLAB for rapid prototyping. Next they converted the model into C and implemented it using fixed-point arithmetic. Lastly they converted the C model to VHDL.

Pyha, developed in the process of this thesis project, aims to aims to bring all the development into the Python domain.

Python is especially suitable for writing the model, rapid prototyping and testing code. Lately Python has been gaining traction over MATLAB in scinetific world, even full resource groups are transitioning *[19]*. Python offers most of what MATLAB has, for example Numpy for numerical computing and Scipy. Matplotlib for figures. In domain of communication systems all the GNURadio blocks have Python mappings. Reproducible research, data and ML.

This thesis proposes an model based design with test-driven approach. Designing the model in Python language is definetly easier considering there are now many libraries that can be used. Pyha includes functions that help verification by automatically running all the simulations, asserting that model is equvalent to the synthesis result, tests defined for model can be reused for RTL, model based verificaiton. Pyha designs are also simulatable and debuggable in Python domain. Pyha also provides an fixed-point type with semi-automatic conversion to it from the floating point values. The design of Pyha also supports fully automatic conversion but currently this is left as a future work.

One major advantage of Pyha is that existing blocks can be connected together in purely Pythonic way, the designer needs to know nothing about hardware design or underlying RTL implementation.

Pyha aims to raise the abstraction level by embracing the object-oriented style. That gives full power of RTL design and good way to abstract away the complexity. Thing that makes Pyha special is that it is an fully sequential language, which would classify it in the HLS category. conclude that BlueSpec is at an intermediate abstraction level between a high level design language (e.g. C) and RTL. Because of this, BlueSpec can handle both data-oriented and control-oriented applications well. [daes]

One of the strentgths of this tool is that it converts to VHDL very simply. That is possible as the synthesis tools are already capable of elobarating (combinatory) sequnetial VHDL code. This thesis contributes the object-oriented VHDL desing way that allows defining registers in sequential code. Thanks to that, the OOP Python code can be simply converet to OOP VHDL code. This is big difference to HLS methods that have to go trough black magic to synthesise the design. This may provide a bridge for VHLD ppl to move to Pyha.

Not limited to this. Long term goal of this project is to develop enough blocks that match the performance of GNURadio, so that flow-grapsh could be simply converted to FPGA designs.

1.3 Structure

First chapter of this thesis gives an overview of the developed tool Pyha and how it can be used for hardware design. Follows the examples that show how Pyha can be used to relatively easly construct moving-average filter and by reusing it the DC-removal filter. Final chapter describes the one of the contribtutions of this thesis, the sequential VHDL OOP model and how Python is converted to it.

Chapter 2

Hardware design with Pyha

This chapter introduces the main contribution of this thesis, Pyha - a tool to design digital hardware in Python.

Pyha proposes to program hardware in the same way as software; much of this chapter is focused on showing differences between hardware and software constructs.

The fist half of the chapter demonstrates how basic hardware constructs can be defined, using Pyha.

The second half introduce the fixed-point type and provides use-cases on designing with Pyha.

All the examples presented in this chapter can be found online HERE, including all the Python sources, unit-tests, VHDL conversion files and Quartus project for synthesis.

Todo

organise examples to web and put link

Todo

write not on how to read examples and simulation results in this chapter!

2.1 Introduction

In this work, Pyha has been designed to follow the object-oriented design paradigm, while many of the other HLS languages work on 'function' based designs. Advantage of the object-oriented way is that the class functions can represent the combinatory logic while class variables represent state..ie registers. This is also more similiar to regular software programming.

Designs in Pyha are fully sequential

KEY IDEA in Pyha is to only add the register behaviour to the

Pyha proposes to use classes as a way of describing hardware. More specifically all the class variables are to be interpreted as hardware registers, this fits well as they are long term state elements.

Todo

improve me

For illustration purposes, Listing 2.1 shows the Pyhe implementation of an adder circuit.

Listing 2.1: Simple adder, implemented in Pyha; `main` is the synthesizable function.

```
class Adder(HW):
    def __init__(self, coef)
        self.coef = coef

    def main(self, x):
        y = x + self.coef
        return y
```

The class structure in Pyha has been designed so that the `__init__` function shall define all the memory elements in the design, it may contain any Python code to evaluate reset values for registers; itself it is not converted to VHDL, only the created variables are interpreted as memory. Class may contain any other user defined functions, that are converted to VHDL; the `main` function is reserved for the top level entity.

Todo

about model! ref blade rf absd? need this because simulations have model output

Note: All the examples in this chapter include the model implementation. In order to keep code examples smaller, future listings omit the model code.

2.1.1 Simulation and testing

One of the motivation in designing the Pyha tool has been the need to improve the verification and testing capabilities. Also verification against an model, consider GNURadio model for example.

Pyha designs can be simulated in Python or VHDL domain. In addition, Pyha has integration to Intel Quartus software, it supports running GATE level simulations i.e. simulation of synthesized logic.

Pyha provides functions to automatically run all the simulations on the set of input data. Listing 2.2 shows an example unit test for the 'adder' module.

Listing 2.2: Unit test for the adder module

```
x =      [1, 2, 2, 3, 3, 1, 1]
expect = [2, 3, 3, 4, 4, 2, 2]

dut = Adder(coef=1)
assert_simulation(dut, expect, x)
```

The `assert_simulation(dut, expect, x)` runs all the simulations (Model, Pyha, RTL and GATE) and asserts the results equal the `expexct` vector, defined in the unit test.

In addition, `simulations(dut, x)` returns all the outputs of different simulations, this can be used to plot the results, as shown in Fig. 2.1.

Todo

remove the input signal from this plot!

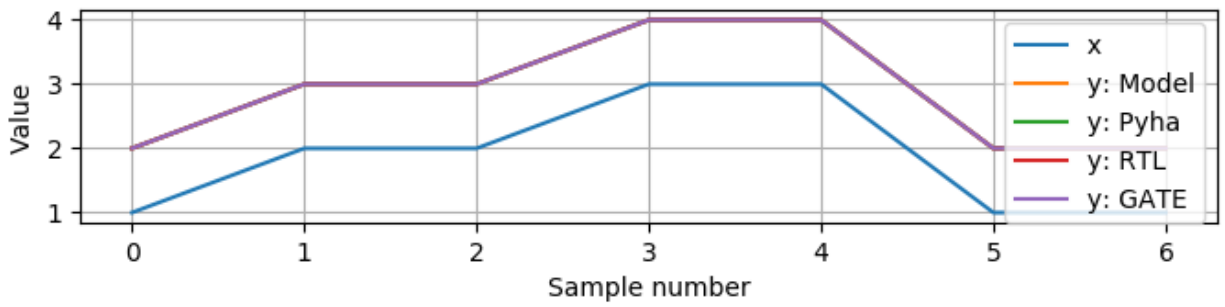

Fig. 2.1: Testing the adder module, `simulation(dut, expect, x)` outputs, all equivalent

More information about the simulation functions can be found in the APPENDIX.

Todo

Add simulation function definitins to appendix.

2.1.2 Synthesis

Synthesis is required to run the GATE level simulations; Pyha integrates to the Intel Quartus software in order to archive this.

Todo

reference blade and lime

Examples in this work use synthesis target device EP4CE40F23C8N, of the Cyclone IV family. This is the same FPGA that powers the latest LimeSDR chip and the BladeRF board. In general it is a low cost FPGA with following features *[20]*:

- 39,600 logic elements;
- 1,134Kbits embedded memory;
- 116 embedded 18x18 multipliers;
- 4 PLLs;
- 200 MHz maximum clock speed.

One useful tool in Quartus software is the RTL viewer, it visualizes the synthesised hardware for the Pyha design, this chapter uses it extensively to illustrate synthesis results.

Fig. 2.2 shows the synthesised RTL diagram of the adder circuit. Notice that the integer types were synthesised to 32 bit logic ([31..0] is the signal width).

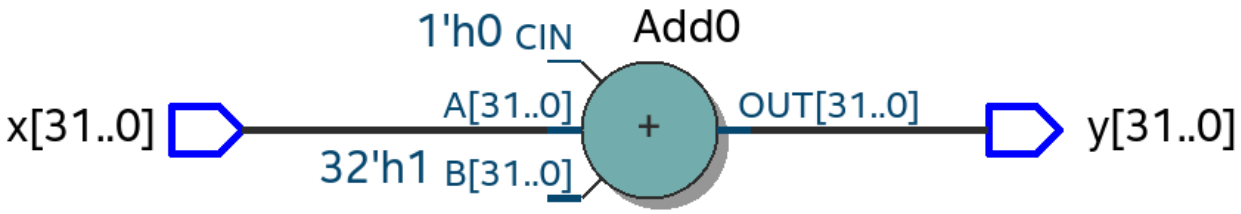

Fig. 2.2: Synthesised RTL of the `Adder(coef=1)` module, `32'1h1` means 32 bit constant with value 1 (Intel Quartus RTL viewer)

2.2 Stateless designs

Todo

improve this, show how functions with only inputs are stateless!

Designs that do not contain any memory elements can be considered stateless (a.k.a. combinatory logic in hardware terms). In the software world, this can be understood as a function that only uses local variables.

2.2.1 Basic operations

Listing 2.3 shows the Pyha design, featuring a circuit with one input and two outputs. Note that the `b` output is dependent on `a`.

Listing 2.3: Basic stateless design with one input `x` and two outputs `a` and `b`

```
class Basic(HW):
    def main(self, x):
        a = x + 1 + 3
        b = a * 314
        return a, b
```

Fig. 2.3 shows that each `+` instruction is mapped to an FPGA resource. The `a` output is formed by adding '1' and '3' to the `x` input. The `b` output has a multiplier on signal path, as expected.

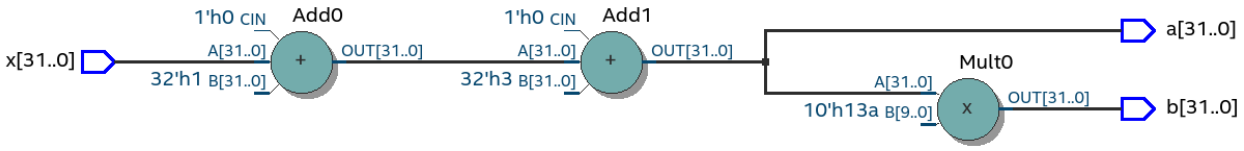

Fig. 2.3: Synthesis result of Listing 2.3 (Intel Quartus RTL viewer)

This example shows that in hardware, operations have a price in terms of resource usage[1]. This is a major difference to software, where operations mainly cost execution time instead.

The key idea to understand is that while the software and hardware execute the `main` function in different ways, they result in the same output, in that sense they are equivalent. This idea is confirmed by Pyha simulation, reporting equal outputs for all simulations, that have been considered in this thesis.

A major advantage of Pyha is that designs can be debugged in Python domain. Pyha simulations just runs the `main` function so all kinds of Python tools can be used. Fig. 2.4 shows a debugging session on the Listing 2.3 code. Using Python tools for debugging can greatly increase the designers productivity.

Todo

[1] Logic elements also introduce delay, but by pipelining this can be neglated.?

more info, this is strong point in this work!

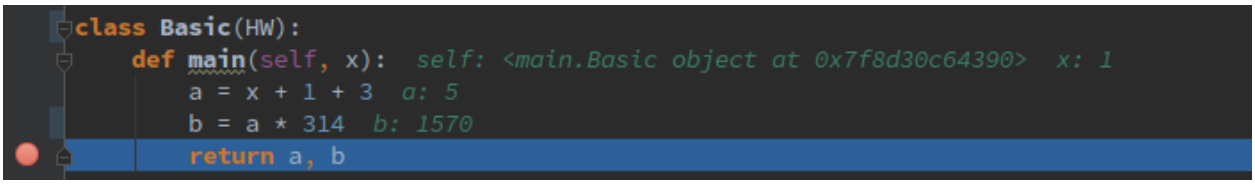

Fig. 2.4: Debugging using PyCharm (Python editor)

2.2.2 Conditional statements

The main conditional statement in Python is `if`, it can be combined with `elif` and `else`. All of these are synthesizable to hardware. Listing 2.4 shows an example of a basic `if else` statement.

Listing 2.4: Example of a basic if else statement in Pyha

```
class If(HW):
    def main(self, x, condition):
        if condition == 0:
            y = x + 3
        else:
            y = x + 1
        return y
```

Fig. 2.5 shows that in hardware the `if` clause is implemented by the 'multiplexer' component. It routes one of the inputs to the output, depending on the value of the condition. For example if `condition == 0` then bottom signal path is routed to output. Interesting thing to note is that both of the adders are 'executing', just one of the result is thrown away.

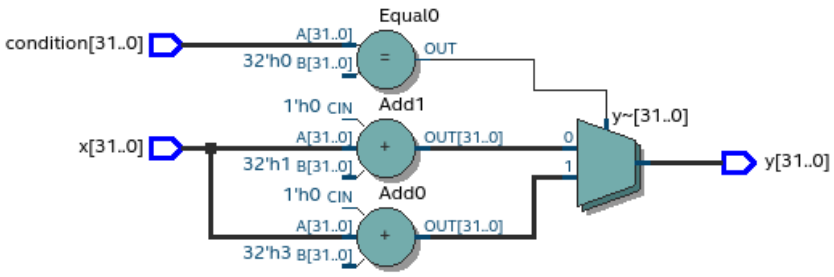

Fig. 2.5: Synthesis result of Listing 2.4 (Intel Quartus RTL viewer)

Once again, all the simulations result in equal outputs. As compared to???

2.2.3 Loop statements

In traditional HDL languages, for loops are usable only for unrolling purposes. Some advanced HLS languages, like for example Vivado HLS, support more complex loops by interpreting them as state machines.

Pyha aims to support the advanced usage of loops, but this is considered as an future work. Currently traditional unrollable loops are supported. This also means that the loop control statement cannot be dynamic.

Listing 2.5 shows an simple `for` example, that adds [0, 1, 2, 3] to the input signal.

Listing 2.5: `for` example

```
class For(HW):
    def main(self, x):
        y = x
        for i in range(4):
            y = y + i

        return y
```

Fig. 2.6 shows that RTL consists of chained adders, that have been also somewhat optimized.

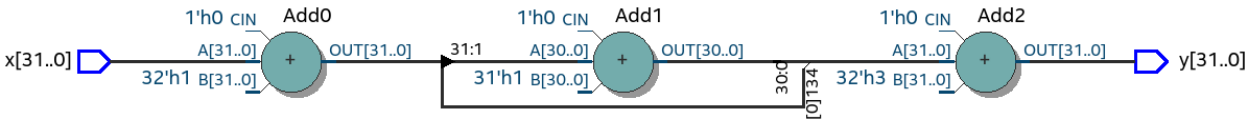

Fig. 2.6: Synthesis result of Listing 2.5 (Intel Quartus RTL viewer)

Todo

this RTL sucks...can we get better example? Maybe combine with if?

The RTL may make more sense if we consider the unrolled version, shown on Listing 2.6.

Listing 2.6: Unrolled `for`, equivalent to Listing 2.5

```
y = x
y = y + 0
y = y + 1
y = y + 2
y = y + 3
```

Most importantly, all the simulations provide equal results.

2.2.4 Function calls

So far only the `main` function has been used to define logic. In Pyha the `main` function is just the top level function that is first called by simulation and conversion processes. Other functions can freely be defined and called as shown in Listing 2.7.

Listing 2.7: Calling an function in Pyha

```
class Functions(HW):
    def adder(self, x, b):
        y = x + b
        return y

    def main(self, x):
        y = self.adder(x, 1)
        return y
```

The synthesis result of Listing 2.7 is just an adder, there is no indication that a function call has been used i.e. all functions are inlined during the synthesis process.

Note that calling the function multiple times would infer parallel hardware.

Todo

maybe i can combine everything in this chapter to one chapter? For example suchs and so the function exampel..

2.2.5 Summary

This chapter demonstrated that many of the software world constructs can be mapped to hardware when expressed in Pyha and that the outputs of the software and hardware simulations are equivalent. Some limitations exist, for example the `for` loops must be unrollable.

Major point to remember is that every statement converted to hardware costs resources on the FPGA fabric.

2.3 Designs with memory

Todo

more general info about the state!

So far, all the designs presented have been stateless (without memory). Often algorithms need to store some value for later use, this indicates that the design must contain memory elements.

This chapter gives an overview of memory based designs in Pyha.

How is this done in Pyha?

2.3.1 Accumulator and registers

Consider the design of an accumulator; it operates by sequentially adding up all the input values. Listing 2.8 shows the Pyha implementation, class scope variable is defined in the `__init__` function to store the accumulator value.

Listing 2.8: Accumulator implemented in Pyha

```
1 class Acc(HW):
2     def __init__(self):
3         self.acc = 0
4
5     def main(self, x):
6         self.acc = self.acc + x
7         return self.acc
```

Trying to run this would result in a Pyha error, suggesting to change the line 6 to `self.next.acc = ...`. After this, the code is runnable; reasons for this modification are explained shortly.

Todo

this is not new, same semantics used in MyHDL and Pong Chu.

The synthesis results shown in the Fig. 2.7 features an new element known as a register.

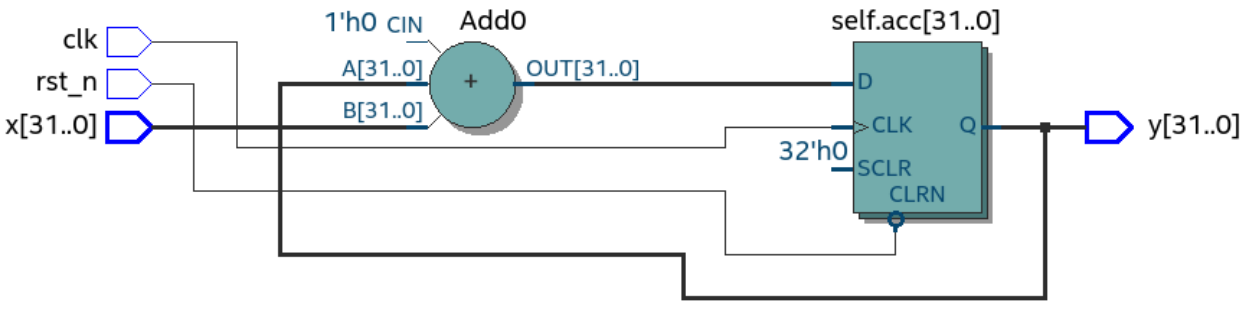

Fig. 2.7: Synthesis result of Listing 2.5 (Intel Quartus RTL viewer)

Register

Todo

this section is not finished

In software programming, class variables are the main method of saving the some information from function call to another.

A register is a hardware memory component; it samples the input signal `D` on the edge of the `CLK` signal. In that sense it acts like a buffer.

One of the new signals in the Fig. 2.7 is `clk`, that is a clock signal that instructs the registers to update the saved value (`D`).

In hardware a clock is a mean of synchronizing the registers, thus allowing accurate timing analsys that allows placing the components on the FPGA fabric in such way that all the analog transients happen **between** the clock edges, thus the registers are guaranteed to sample the clean and correct signals.

Registers have one difference to software class variables i.e. the value assigned to them does not take effect immediately, but rather on the next clock edge. When the value is set at **this** clock edge, it will be taken on the **next** clock edge.

Pyha tries to stay in the software world, so the clock signal can be abstracted away by thinking that it denotes the call to the 'main' function. This means that registers update their value on every call to `main` (just before the call).

Think that the `main` function is started with the **current** register values known and the objective of the `main` function is to find the **next** values for the registers.

Todo

This sample rate stuff is too bold statement, more expalnation

Furthermore, in DSP systems one important aspect is sample rate. In hardware the maximum clock rate and sample rate are basically the same thing. In Digital signal processing applications we have sampling rate, that is basically equal to the clock rate. Think that for each input sample the 'main' function is called, that is for each sample the clock ticks.

Note that the way how the hardware is designed determines the maximum clock rate it can run off. So if we do a bad job we may have to work with low sample rate designs. This is determined by the worst critical path.

The Pyha way is to register all the outputs, that way i can be assured that all the inputs are already registered.

The `rst_n` signal can be used to set initial states for registers, in Pyha the initial value is determined by the value assigned in `__init__`, in this case it is 0.

Testing

Simulation results in Fig. 2.8 show that the **model** simulation differs from the rest of the simulations. It is visible that the hardware related simulations are **delayed by 1** sample. This is the side-effect of the hardware registers, each register on the signal path adds one sample delay.

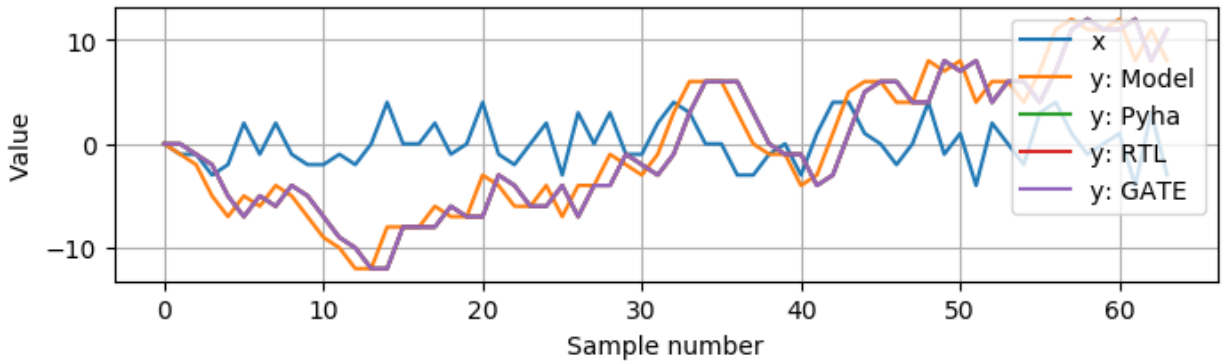

Fig. 2.8: Simulation of the accumulator (x is a random integer [-5;5])

Pyha provides a `self._delay` variable, that hardware classes can use to specify their delay. Simulation functions can read this variable and compensate the simulation data so that the delay is compensated, that eases the design of unit-tests.

The simulation results match in output (Fig. 2.9), after setting the `self._delay = 1` in the `__init__` function.

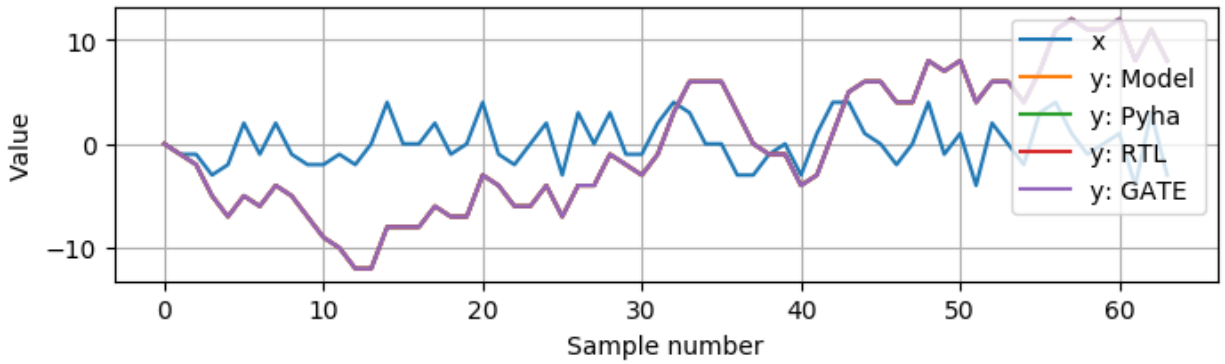

Fig. 2.9: Simulation of the delay-compensated accumulator (x is a random integer [-5;5])

2.3.2 Block processing and sliding adder

One very common task in DSP designs is to calculate results based on some number of input samples (block processing). Until now, the `main` function has worked with a single input sample, this can now be changed by keeping the history with registers.

Consider an algorithm that adds the last 4 input values. Listing 2.9 shows an implementation that keeps track of the last 4 input values and sums them. Note that the design also uses the output register `y`.

Listing 2.9: Sliding adder algorithm

```
class SlidingAdder(HW):
    def __init__(self):
        self.shr = [0, 0, 0, 0] # list of registers
        self.y = 0

    def main(self, x):
        # add new 'x' to list, throw away last element
        self.next.shr = [x] + self.shr[:-1]

        # add all element in the list
        sum = 0
        for a in self.shr:
            sum = sum + a

        self.next.y = sum
        return self.y
```

The `self.next.shr = [x] + self.shr[:-1]` line is also known as a 'shift register', because on every call it shifts the list contents to the right and adds new `x` as the first element. Sometimes the same structure is used as a delay-chain, because the sample `x` takes 4 updates to travel from `shr[0]` to `shr[3]`. This is a very common element in hardware DSP designs.

Fig. 2.10 shows the RTL for this design, as expected the `for` has been unrolled, thus all the summing is done.

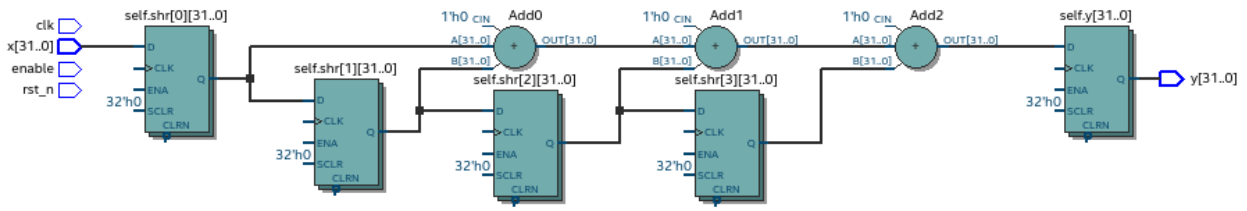

Fig. 2.10: Synthesis result of Listing 2.9 (Intel Quartus RTL viewer)

Optimizing the design

This design can be made generic by changing the `__init__` function to take the window length as a parameter (Listing 2.10).

Listing 2.10: Generic sliding adder

```
class SlidingAdder(HW):
    def __init__(self, window_len):
        self.shr = [0] * window_len
    ...
```

The problem with this design is that it starts using more resources as the `window_len` gets larger as every stage requires a separate adder. Another problem is that the critical path gets longer, decreasing the clock rate. For example, the design with `window_len=4` synthesises to maximum clock of 170 MHz, while `window_len=6` to only 120 MHz.

Todo

MHz on what FPGA?

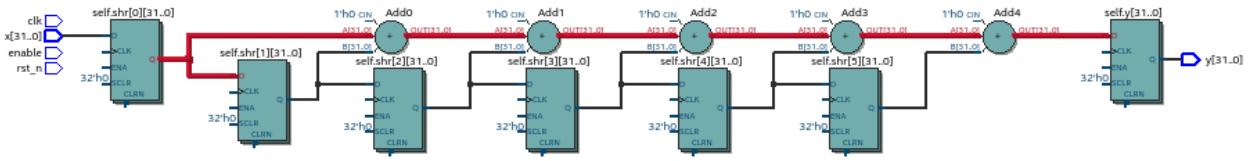

Fig. 2.11: RTL of `window_len=6`, the red line shows the critical path (Intel Quartus RTL viewer)

In that sense, it can be considered a poor design, as it is hard to reuse. Conveniently, the algorithm can be optimized to use only 2 adders, no matter the window length. Listing 2.11 shows that instead of summing all the elements, the overlapping part of the previous calculation can be used to significantly optimize the algorithm.

Listing 2.11: Optimizing the sliding adder algorithm by using recursive implementation

```
y[4] = x[4] + x[5] + x[6] + x[7] + x[8] + x[9]
y[5] =        x[5] + x[6] + x[7] + x[8] + x[9] + x[10]
y[6] =               x[6] + x[7] + x[8] + x[9] + x[10] + x[11]

# reusing overlapping parts implementation
y[5] = y[4] + x[10] - x[4]
y[6] = y[5] + x[11] - x[5]
```

Listing 2.12 gives the implementation of the optimal sliding adder; it features a new reg-

ister `sum`, that keeps track of the previous output. Note that the ``shr` stayed the same, but is now rather used as a delay-chain.

Listing 2.12: Optimal sliding adder

```
class OptimalSlideAdd(HW):
    def __init__(self, window_len):
        self.shr = [0] * window_len
        self.sum = 0

        self._delay = 1

    def main(self, x):
        self.next.shr = [x] + self.shr[:-1]

        self.next.sum = self.sum + x - self.shr[-1]
        return self.sum
    ...
```

Fig. 2.12 shows the synthesis result; as expected, the critical path is along 2 adders.

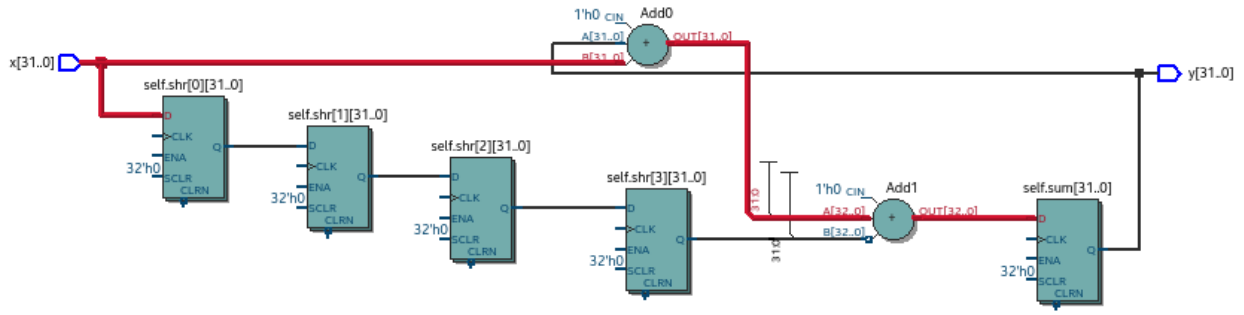

Fig. 2.12: Synthesis result of Listing 2.9, `window_len=4` (Intel Quartus RTL viewer)

Simulations results (Fig. 2.13) show that the hardware desing behaves exactly as the software model. Note that the class has `self._delay=1` to compensate for the register delay.

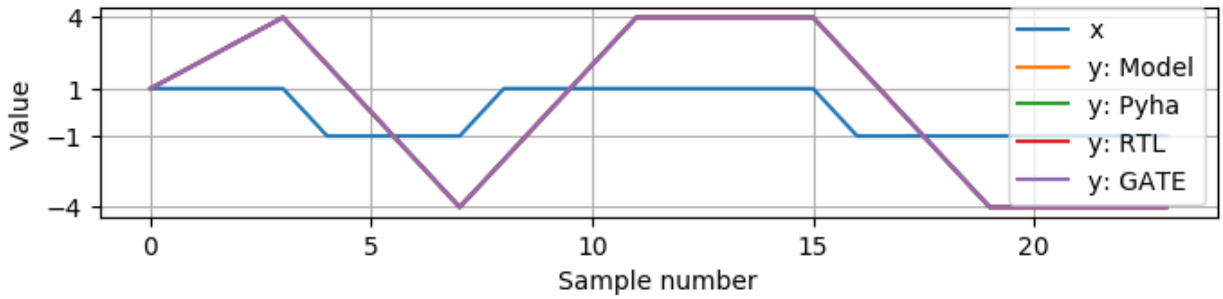

Fig. 2.13: Simulation results for `OptimalSlideAdd(window_len=4)`

2.3.3 Summary

In Pyha all class variables are interpreted as hardware registers. The `__init__` function may contain any Python code to evaluate reset values for registers.

The key difference between software and hardware approaches is that hardware registers have **delayed assignment**, they must be assigned to `self.next`.

The delay introduced by the registers may drastically change the algorithm, that is why it is important to always have a model and unit tests, before starting hardware implementation. The model delay can be specified by `self._delay` attribute, this helps the simulation functions to compensate for the delay.

Registers are also used to shorten the critical path of chained logic elements, thus allowing higher clock rate. It is encouraged to register all the outputs of Pyha designs.

2.4 Fixed-point designs

Examples in the previous chapters have used only the `integer` type, in order to simplify the designs.

Todo

explain why float costs greatly?

DSP algorithms are mostly described using floating point numbers. As shown in previous sections, every operation in hardware takes resources and floating point calculations cost greatly. For that reason, fixed-point arithmetic is often used in hardware designs.

Fixed-point arithmetic is in nature equal to integer arithmetic and thus can use the DSP blocks that come with many FPGAs (some high-end FPGAs have also floating point DSP blocks *[21]*).

2.4.1 Basics

Pyha defines `Sfix` for FP objects; it is a signed number. It works by defining bits designated for `left` and `right` of the decimal point. For example `Sfix(0.3424, left=0, right=-17)` has 0 bits for integer part and 17 bits for the fractional part. Listing 2.13 shows some examples. more information about the fixed point type is given on APPENDIX.

Todo

Add more information about fixed point stuff to the appendix

Listing 2.13: Example of `Sfix` type, more bits give better results

```
>>> Sfix(0.3424, left=0, right=-17)
0.34239959716796875 [0:-17]
>>> Sfix(0.3424, left=0, right=-7)
0.34375 [0:-7]
>>> Sfix(0.3424, left=0, right=-4)
0.3125 [0:-4]
```

The default FP type in Pyha is `Sfix(left=0, right=-17)`, it represents numbers between [-1;1] with resolution of 0.000007629. This format is chosen because it fits into common FPGA DPS blocks (18 bit signals *[20]*) and it can represent normalized numbers.

The general recommendation is to keep all the inputs and outputs of the block in the default type.

2.4.2 Fixed-point sliding adder

Consider converting the sliding window adder, described in Section 2.3.2, to FP implementation. This requires changes only in the `__init__` function (Listing 2.14).

Listing 2.14: Fixed-point sliding adder

```
def __init__(self, window_size):
    self.shr = [Sfix()] * window_size
    self.sum = Sfix(left=0)
...
```

The first line sets `self.shr` to store `Sfix()` elements. Notice that it does not define the fixed-point bounds, meaning it will store 'whatever' is assigned to it. The final bounds are determined during simulation.

Todo

lazy stuff needs more explanation

The `self.sum` register uses another lazy statement of `Sfix(left=0)`, meaning that the integer bits are forced to 0 bits on every assign to this register. The fractional part is left determined by simulation. The rest of the code is identical to the one described in Section 2.3.2.

Synthesis results are shown in Fig. 2.14. In general, the RTL diagram looks similar to the one at Section 2.3.2. First noticeable change is that the signals are now 18 bits wide due

to the default FP type. The second addition is the saturation logic, which prevents the wraparound behaviour by forcing the maximum or negative value when they are out of fixed point format. Saturation logic is by default enabled for FP types.

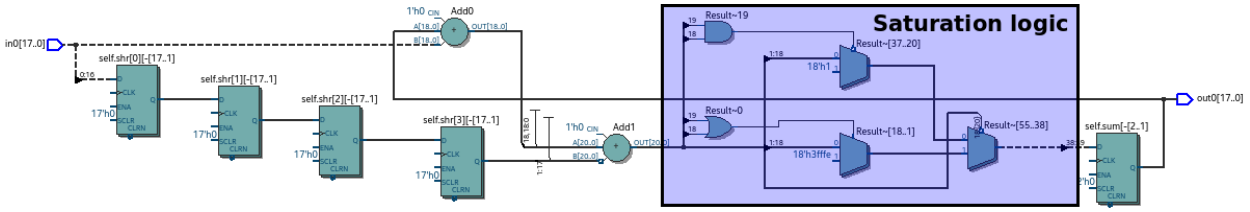

Fig. 2.14: RTL of fixed-point sliding adder (Intel Quartus RTL viewer)

Fig. 2.15 plots the simulation results for input of random signal in [-0.5;0.5] range. Notice that the hardware simulations are bounded to [-1;1] range by the saturation logic, that is why the model simulation is different at some points.

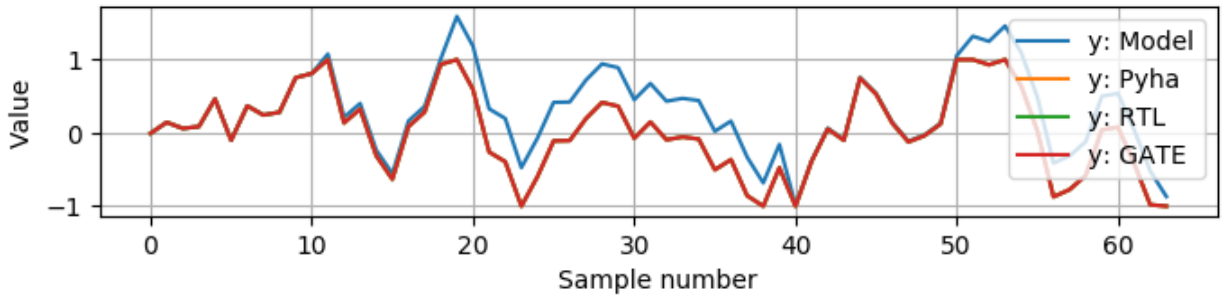

Fig. 2.15: Simulation results of FP sliding sum

Simulation functions can automatically convert 'floating-point' inputs to default FP type. In same manner, FP outputs are converted to floating point numbers. That way, the designer does not have to deal with FP numbers in unit-testing code. An example is given in Listing 2.15.

Listing 2.15: Test fixed-point design with floating-point numbers

```
dut = OptimalSlidingAddFix(window_len=4)
x = np.random.uniform(-0.5, 0.5, 64)
y = simulate(dut, x)
# plotting code ...
```

2.4.3 Moving average filter

Todo

rephrase as this is copy paste!

The moving average (MA) 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 *[22]*.

Fig. 2.16 shows that MA is a good algorithm for noise reduction. Increasing the window length reduces more noise but also increases the complexity and delay of the system (MA is a special case of FIR filter, same delay semantics apply).

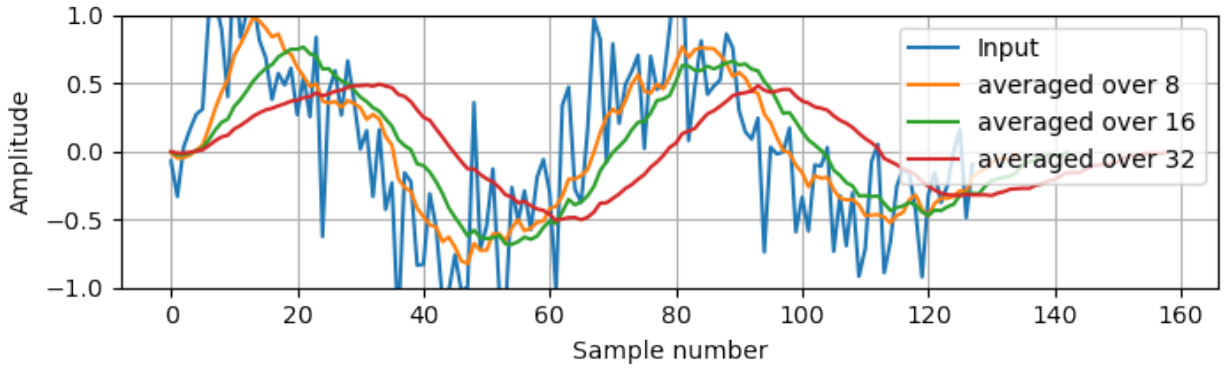

Fig. 2.16: MA algorithm in removing noise

Good noise reduction performance can be explained by the frequency response of MA (Fig. 2.17), showing that it is a low-pass filter. Passband width and stopband attenuation are controlled by the window length.

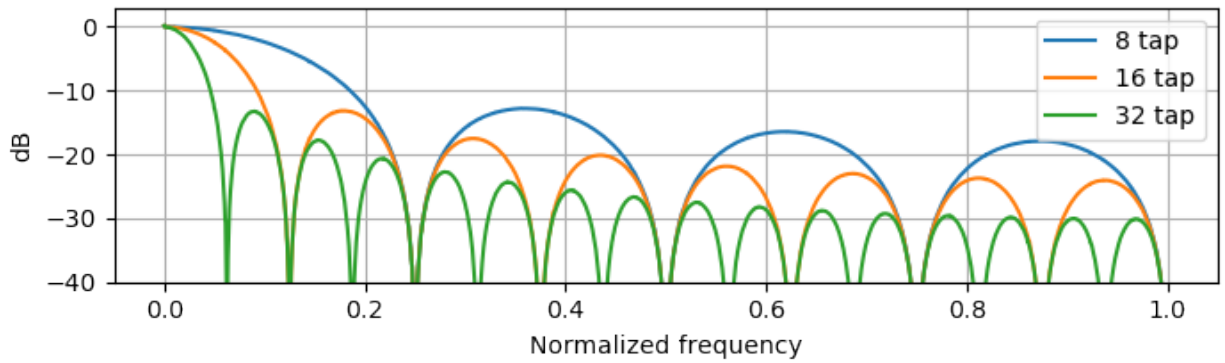

Fig. 2.17: Frequency response of MA filter

Implementation in Pyha

MA is implemented by using a sliding sum that is divided by the sliding window length. The sliding sum part has already been implemented in Section 2.4.2. The division can be implemented by a shift right operation if the divisor is power of two.

In addition, division can be performed on each sample instead of on the sum, that is `(a + b) / c = a/c + b/c`. Doing this guarantees that the `sum` variable is always in the [-1;1] range, thus the saturation logic can be removed.

Listing 2.16: MA implementation in Pyha

```
1  class MovingAverage(HW):
2      def __init__(self, window_len):
3          self.window_pow = Const(int(np.log2(window_len)))
4
5          self.mem = [Sfix()] * window_len
6          self.sum = Sfix(0, 0, -17, overflow_style=fixed_wrap)
7          self._delay = 1
8
9      def main(self, x):
10         div = x >> self.window_pow
11
12         self.next.mem = [div] + self.mem[:-1]
13         self.next.sum = self.sum + div - self.mem[-1]
14         return self.sum
15     ...
```

The code in Listing 2.16 makes only few changes to the sliding sum:

- On line 3, `self.window_pow` stores the bit shift count (to support generic `window_len`)
- On line 6, type of `sum` is changed so that saturation is turned off
- On line 10, shift operator performs the division

Fig. 2.18 shows the synthesized result of this work; as expected it looks very similar to the sliding sum RTL schematics. In general, shift operators are hard to notice on the RTL schematics because they are implemented by routing semantics.

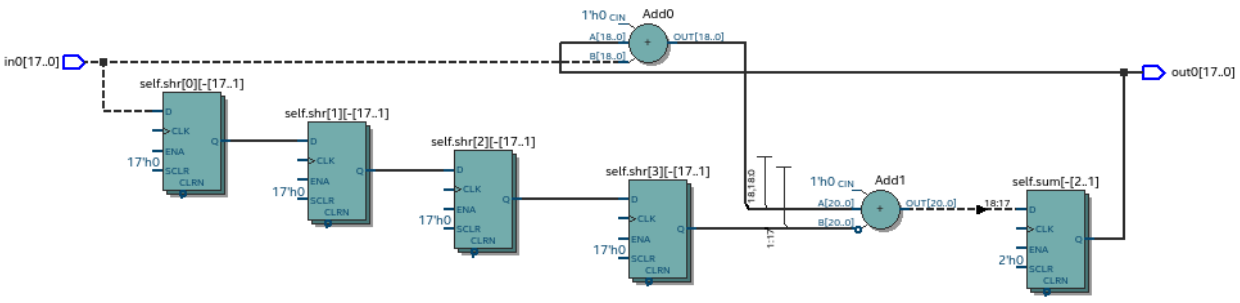

Fig. 2.18: RTL view of moving average (Intel Quartus RTL viewer)

Simulation/Testing

MA is an optimal solution for performing matched filtering of rectangular pulses *[22]*. This is important for communication systems. Fig. 2.19 shows an example of and digital signal, that is corrupted with noise. MA with window length equal to samples per symbol can recover the signal from the noise.

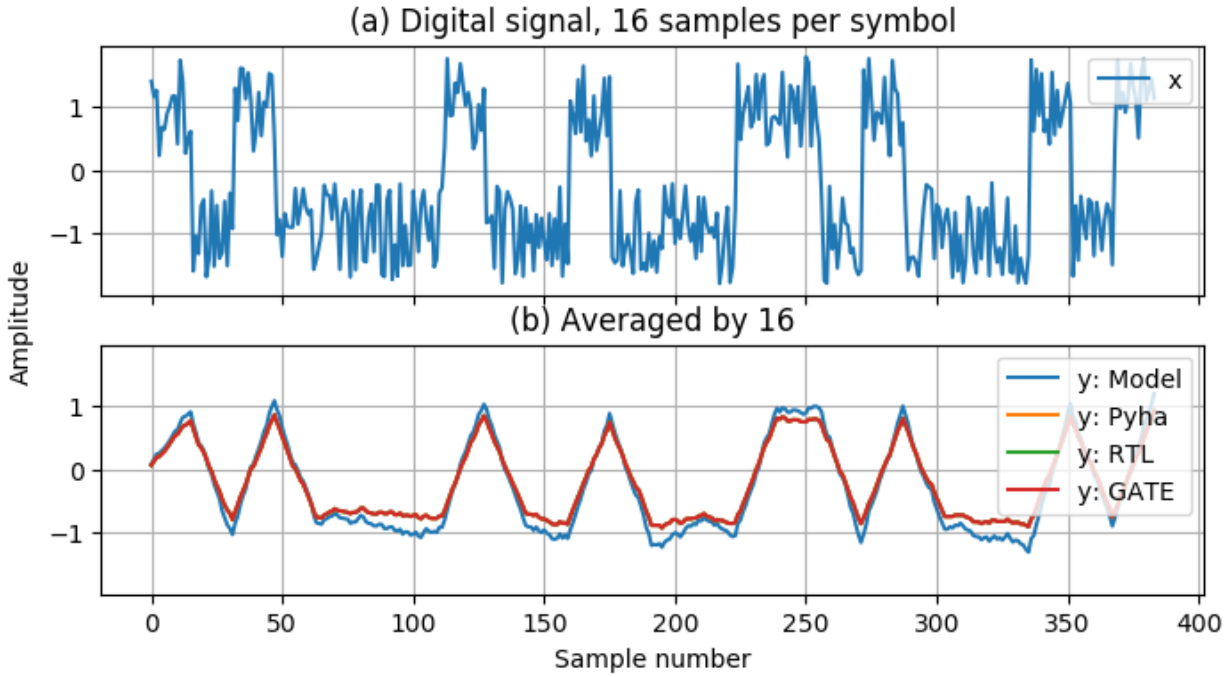

Fig. 2.19: Moving average as matched filter

The 'model' deviates from rest of the simulations because the input signal violates the [-1;1] bounds and hardware simulations are forced to saturate the values.

2.4.4 Summary

In Pyha, DSP systems can be implemented by using the fixed-point type. The combination of 'lazy' bounds and default Sfix type provide simplified conversion from floating point to fixed point. In that sense it could be called 'semi-automatic conversion'.

Simulation functions can automatically perform the floating to fixed point conversion, this enables writing unit-tests using floating point numbers.

Comparing the FP implementation to the floating-point model can greatly simplify the final design process.

2.5 Abstraction and Design reuse

Pyha has been designed in the way that it can represent RTL designs exactly as the user defines, however thanks to the object-oriented nature all these low level details can be abstracted away and then Pyha turns into HLS language. To increase productivity, abstraction is needed.

Pyha is based on the object-oriented design practices, this greatly simplifies the design reuse as the classes can be used to initiate objects. Another benefit is that classes can abstract away the implementation details, in that sense Pyha can become a high-level synthesis (HLS) language.

This chapter gives an example on how to reuse the moving average filter for ...

2.5.1 Linear-phase DC removal Filter

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 *[23]*.

DC offset looks like a peak near the 0Hz on the frequency response. In the time domain, it manifests as a constant component on the harmonic signal.

In *[24]*, Rick Lyons investigates the use of moving average algorithm as a DC removal circuit. This works by subtracting the MA output from the input signal. The problem of this approach is the 3 dB passband ripple. However, by connecting multiple stages of MA's in series, the ripple can be avoided (Fig. 2.20) *[24]*.

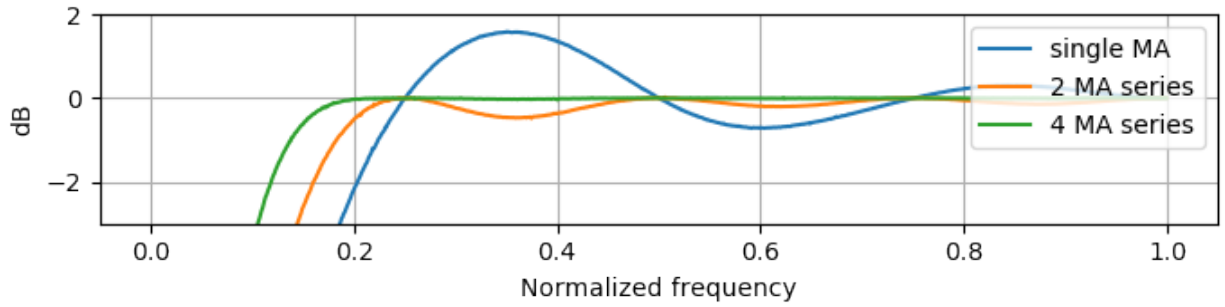

Fig. 2.20: Frequency response of DC removal filter (MA window length is 8)

Implementation

The algorithm is composed of two parts. First, four MA's are connected in series, outputting the DC component of the signal. Second, the MA's output is subtracted from the input signal, thus giving the signal without DC component. Listing 2.17 shows the Pyha implementation.

Listing 2.17: DC-Removal implementation

```
class DCRemoval(HW):
    def __init__(self, window_len):
        self.mavg = [MovingAverage(window_len), MovingAverage(window_len),
                     MovingAverage(window_len), MovingAverage(window_len)]
        self.y = Sfix(0, 0, -17)

        self._delay = 1

    def main(self, x):
        # run input signal over all the MA's
        tmp = x
        for mav in self.mavg:
            tmp = mav.main(tmp)

        # dc-free signal
        self.next.y = x - tmp
        return self.y
    ...
```

This implementation is not exactly following that of *[24]*. They suggest to delay-match the step 1 and 2 of the algorithm, but since we can assume the DC component to be more or less stable, this can be omitted.

Fig. 2.21 shows that the synthesis generated 4 MA filters that are connected in series, output of this is subtracted from the input.

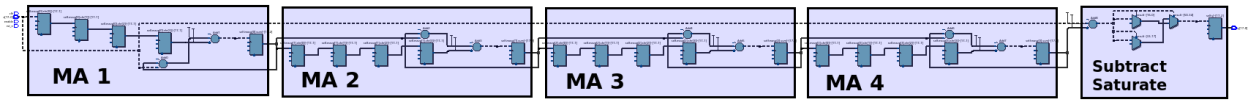

Fig. 2.21: Synthesis result of `DCRemoval(window_len=4)` (Intel Quartus RTL viewer)

In a real application, one would want to use this component with larger `window_len`. Here 4 was chosen to keep the RTL simple. For example, using `window_len=64` gives much better cutoff frequency (Fig. 2.22); FIR filter with the same performance would require hundreds of taps *[24]*. Another benefit is that this filter delays the signal by only 1 sample.

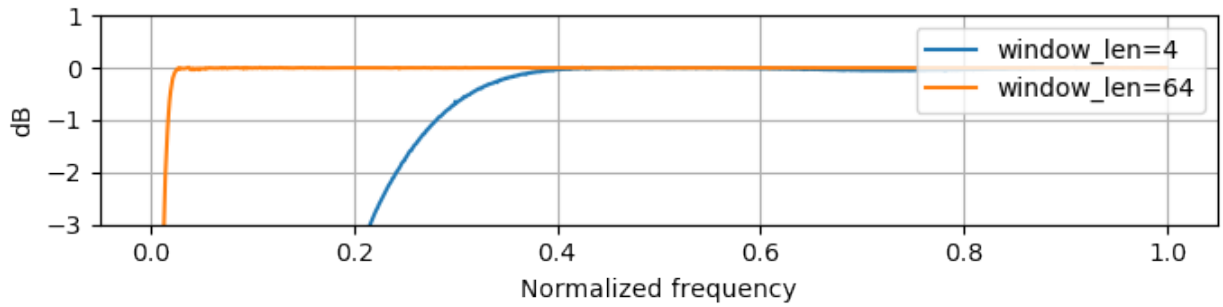

Fig. 2.22: Comparison of frequency response

This implementation is also very light on the FPGA resource usage (Listing 2.18).

Listing 2.18: Cyclone IV FPGA resource usage for `DCRemoval(window_len=64)`

```
Total logic elements                242 / 39,600 ( < 1 % )
Total memory bits                   2,964 / 1,161,216 ( < 1 % )
Embedded Multiplier 9-bit elements  0 / 232 ( 0 % )
```

Testing

Fig. 2.23 shows the situation where the input signal is corrupted with a DC component (+0.25), the output of the filter starts countering the DC component until it is removed.

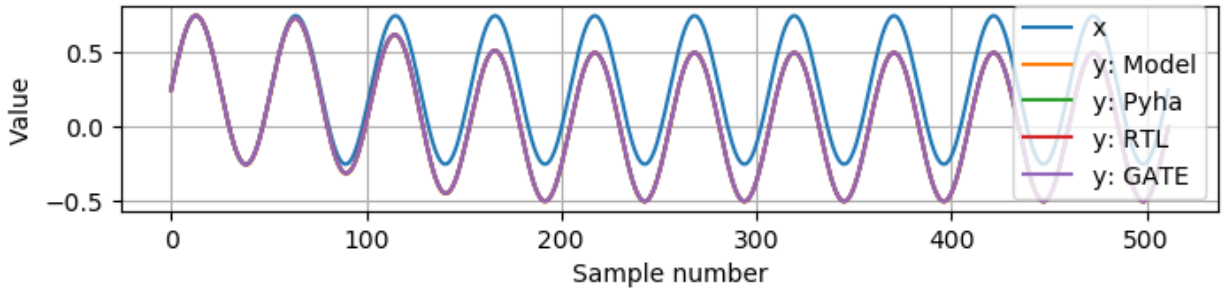

Fig. 2.23: Simulation of DC-removal filter in the time domain

2.6 Conclusion

This chapter has demonstrated that in Pyha traditional software language features can be used to infer hardware components and their outputs are equivalent. One must still keep in mind how the code converts to hardware, for example that the loops will be unrolled. A major difference between hardware and software is that in hardware, every arithmetical operator takes up resources.

Class variables can be used to add memory to the design. In Pyha, class variables must be assigned to `self.next` as this mimics the **delayed** nature of registers. The general rule is to always register the outputs of Pyha designs.

DSP systems can be implemented by using the fixed-point type. Pyha has 'semi-automatic conversion' from floating point to fixed point numbers. Verifying against floating point model helps the design process.

Reusing Pyha designs is easy thanks to the object-oriented style that also works well for design abstraction.

Pyha provides the `simulate` function that can automatically run Model, Pyha, RTL and GATE level simulations. In addition, `assert_simulate` can be used for fast design of unit-

tests. These functions can automatically handle fixed point conversion, so that tests do not have to include fixed point semantics. Pyha designs are debuggable in the Python domain.

Chapter 3

Conversion to VHDL

This chapter shows how Pyha converts to VHDL.

Todo

Pilt converterist

- Simulatsioon tüüpide leidmiseks
- Python syntax to VHDL
- Sequential OOP VHDL IR

First part of this chapter introduces the Sequential OOP VHDL IR.

While other high level tools convert to very low-level VHDL, then Pyha takes and different approach by first developing an feasible model in VHDL and then using Python to get around VHDL ugly parts.

Many tools on the market are capable of converting higher level language to VHDL. However, these tools only make use of the very basic dataflow semantics of VHDL language, resulting in complex conversion process and typically unreadable VHDL output.

3.1 Sequential, Object-oriented style for VHDL

This chapter develops sequential synthesizable object-oriented (OOP) programming model for VHDL. The main motivation is to use it as an intermediate language for High-Level synthesis of hardware.

VHDL has been chosen over SystemVerilog(SV) because it is a strict language and forbids many mistakes during compile time. SV on the other hand is much more permissive, for example allowing out-of-bounds array indexing *[1]*.

Sequential programming in VHDL has been studied by Jiri Gaislter in *[2]*. He showed that combinatory logic is easily described by fully sequential functions. He proposesed the 'two-process' design method, where one of the processes is for comb and other for registers. Hes work is limited to one clock domain.

This sections contribution is the extension of the 'two process' model by adding an Object-oriented approach. The basic idea of OOP is to bundle data and define functions that perform actions on this data. This idea fits well with hardware design, as 'data' can be thought as registers and combinatory logic as functions that perform operations on the data.

VHDL has no direct support, but the OOP style can be still used by by combining data in records (same as 'C' struct) and passing them as a parameters to functions. This is essentially the same way how C programmers do it.

Listing 3.1 demonstrates pipelined multiply-accumuluate(MAC), written in OOP VHDL. Recall that all the items in the `self_t` are to be registers. One inconvenience is that VHDL procedures cannot 'return' , instead 'out' direction arguments must be used. On the other hand this helps to handle Python functions that can return multiple values.

Listing 3.1: OOP style multiply-accumulate in VHDL

```
type self_t is record
    mul: integer;
    acc: integer;
    coef: integer;
end record;

procedure main(self: inout self_t; a: in integer; ret_0: out integer) is
begin
    self.mul := a * self.coef;
    self.acc := self.acc + self.mul;
    ret_0 := self.acc;
end procedure;
```

The synthesis results (Fig. 3.1) show that a functionally correct MAC has been implemented. However, in terms of hardware, it is not quite what was wanted. The data model specified 3 registers, but only the one for 'acc' is present and even this is at the wrong location.

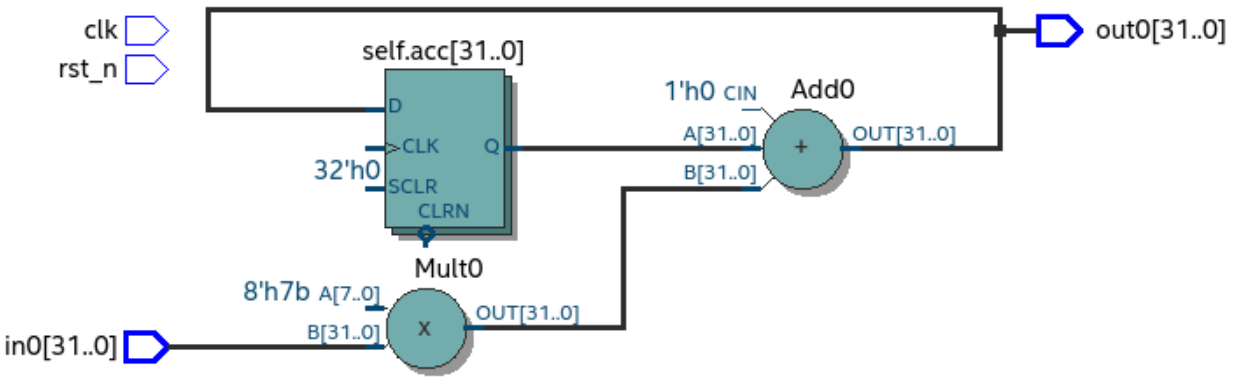

Fig. 3.1: Unexpected synthesis result of Listing 3.1 (Intel Quartus RTL viewer)

3.1.1 Defining registers with variables

Clearly the way of defining registers is not working properly. The mistake was to expect that the registers work in the same way as 'class variables' in traditional programming languages.

Hardware registers have just one difference to class variables, the value assigned to them does not take effect immediately, but rather on the next clock edge. That is the basic idea of registers, they take a new value on clock edge. When the value is set at **this** clock edge, it will be taken on **next** clock edge.

Trying to stay in the software world, we can abstract away the clock edge by thinking that it denotes the call to the 'main' function. Meaning that registers take the assigned value on the next function call, meaning assignment is delayed by one function call.

VHDL defines a special assignment operator for this kind of delayed assignment, it is called 'signal assignment'. It must be used on VHDL signal objects like `a <= b`.

VHDL signals really come down to just having two variables, to represent the **next** and **current** values. Signal assignment operator sets the value of **next** variable. On the next simulation delta, **current** is automatically set to equal **next**.

This two variable method has been used before, for example Pong P. Chu, author of one of the most reputed VHDL books, suggests to use this style in defining sequential logic in VHDL *[3]*. The same semantics are also used in MyHDL signal objects *[4]*.

Adapting this style for the OOP data model is shown in `mac-next-data`. The new data model extends the structure to include the 'nexts' object, that can used to assign **next** value for registers, for example `self.nexts.acc := 0`.

Listing 3.2: Data model with **next**, in OOP-style VHDL

```
type next_t is record
    mul: integer;
    acc: integer;
```

```
    coef: integer;
end record;

type self_t is record
    mul: integer;
    acc: integer;
    coef: integer;

    nexts: next_t; -- new element to hold 'next state' value
end record;

procedure main(self: inout self_t; a: integer; ret_0: out integer) is
begin
    self.nexts.mul := a * self.coef;        -- now assigns to self.nexts
    self.nexts.acc := self.acc + self.mul;  -- now assigns to self.nexts
    ret_0 := self.acc;
end procedure;
```

Now the loading of **next** to **current** must now be done manually. Listing 3.3 defines new function 'update_registers', taking care of this task.

Listing 3.3: Function to update registers, in OOP-style VHDL

```
procedure update_register(self: inout self_t) is
begin
    self.mul := self.nexts.mul;
    self.acc := self.nexts.acc;
    self.coef:= self.nexts.coef;
end procedure;
```

Note: Function 'update_registers' is called on clock raising edge. While the 'main' is called as combinatory function.

Todo

add simple top level example here?

Synthesising this results in expected logic, that is MAC with pipelined registers (Fig. 3.2).

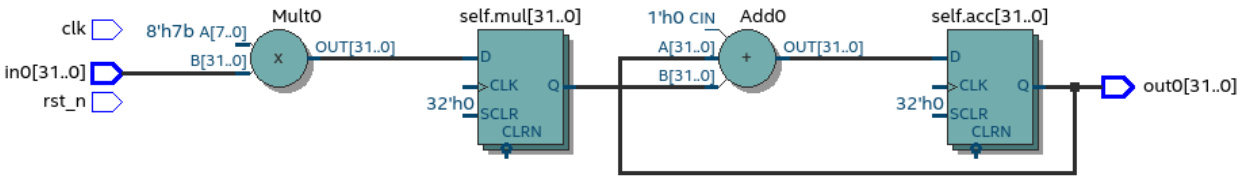

Fig. 3.2: Synthesis result of the revised code (Intel Quartus RTL viewer)

3.1.2 Creating instances

Todo

consider removing this section, quite useless..

The general approach of creating instances is to define new variables of the 'self_t' type, Listing 3.4 gives an example of this.

Listing 3.4: Class instances by defining records, in OOP-style VHDL

```
variable mac0: MAC.self_t;
variable mac1: MAC.self_t;
```

The next step is to initialize the variables, this can be done at the variable definition, for example: `variable mac0: self_t := (mul=>0, acc=>0, coef=>123, nexts=>(mul=>0, acc=>0, coef=>123));`

The problem with this method is that all data-model must be initialized (including 'nexts'), this will get unmaintainable very quickly, imagine having an instance that contains another instance or even array of instances. In some cases it may also be required to run some calculations in order to determine the initial values.

Traditional programming languages solve this problem by defining class constructor, executing automatically for new objects.

In the sense of hardware, this operation can be called 'reset' function. Listing 3.5 is a reset function for the MAC circuit. It sets the initial values for the data model and can also be used when reset signal is asserted.

Listing 3.5: Reset function for MAC, in OOP-style VHDL

```
procedure reset(self: inout self_t) is
begin
    self.nexts.coef := 123;
    self.nexts.mul := 0;
    self.nexts.sum := 0;
```

```
    update_registers(self);
end procedure;
```

But now the problem is that we need to create a new reset function for each instance.

This can be solved by using VHDL 'generic packages' and 'package instantiation declaration' semantics *[5]*. Package in VHDL just groups common declarations to one namespace.

In case of the MAC class, the 'coef' reset value could be set as package generic. Then each new package initialization could define new reset value for it (Listing 3.6).

Listing 3.6: Initialize new package MAC_0, with 'coef' 123

```
package MAC_0 is new MAC
   generic map (COEF => 123);
```

Unfortunately, these advanced language features are not supported by most of the synthesis tools. A workaround is to either use explicit record initialization (as at the start of this chapter) or manually make new package for each instance.

Both of these solutions require unnecessary workload.

The Python to VHDL converter (developed in the next chapter), uses the later option, it is not a problem as everything is automated.

3.1.3 Final OOP model

Currently the OOP model consists of following elements:

- Record for 'next'
- Record for 'self'
- User defined functions (like 'main')
- 'Update registers' function
- 'Reset' function

VHDL supports 'packages' to group common types and functions into one namespace.

Listing 3.7 shows the template package for VHDL 'class'. All the class functionality is now in common namespace.

Listing 3.7: Package template for OOP style VHDL

```
package Class is
    type next_t is record
        ...
    end record;
```

```
    type self_t is record
        ...
        nexts: next_t;
    end record;

    -- function prototypes
end package;

package body Class is
    procedure reset(self: inout self_t) is
        ...
    procedure update_registers(self: inout self_t) is
        ...
    procedure main(self:inout self_t) is
        ...
    -- other user defined functions
end package body;
```

3.1.4 Examples

Creating a new class that connects two MAC instances in series is simple, first we need to create two MAC packages called MAC_0 and MAC_1 and add them to the data model (Listing 3.8). The next step is to call MAC_0 operation on the input and then pass the output trough MAC_1, whose output is the final output (`mac-series-main`).

Todo

why MAC_0 and MAC_1?

Listing 3.8: Series MACs in OOP-style VHDL

```
type self_t is record
    mac0: MAC_0.self_t; -- define 2 MACs as part of data model
    mac1: MAC_1.self_t;

    nexts: next_t;
end record;

procedure main(self:inout self_t; a: integer; ret_0:out integer) is
    variable out_tmp: integer;
begin
    MAC_0.main(self.mac0, a, ret_0=>out_tmp);       -- connect MAC_0 output to␣
↪MAC_1 input
```

```
    MAC_1.main(self.mac1, out_tmp, ret_0=>ret_0);
end procedure;
```

Synthesis result shows that two MACs are connected in series Fig. 3.3.

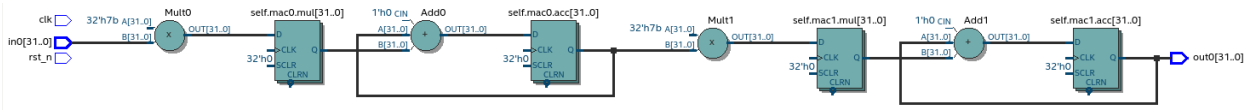

Fig. 3.3: Synthesis result of the new class (Intel Quartus RTL viewer)

Connecting two MAC's instead in parallel can be done with simple modification to 'main' function, that instead now returns both outputs (Listing 3.9).

Listing 3.9: Main function for parallel instances, in OOP-style VHDL

```
procedure main(self:inout self_t; a: integer; ret_0:out integer; ret_1:out␣
↪integer) is
begin
    MAC_0.main(self.mac0, a, ret_0=>ret_0); -- return MAC_0 output
    MAC_1.main(self.mac1, a, ret_0=>ret_1); -- return MAC_1 output
end procedure;
```

Two MAC's are synthesized in parallel, as shown in Fig. 3.4.

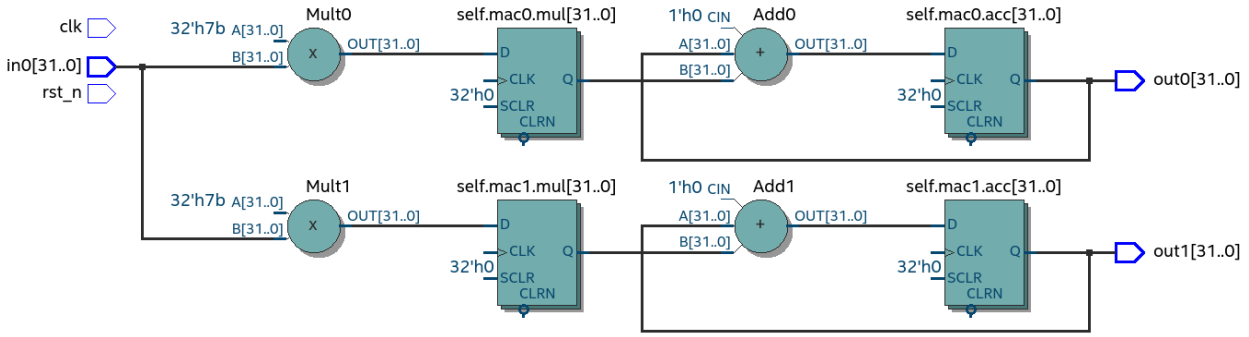

Fig. 3.4: Synthesis result of Listing 3.9 (Intel Quartus RTL viewer)

3.1.5 Multiple clock domains

Multiple clock domains can be easily supported by updating registers at different clock domains. By reusing the parallel MAC's example, consider that MAC_0 and MAC_1 work in different clock domain. For this only the top level process must be modified (Listing 3.10), rest of the code stays the same.

Listing 3.10: Top-level for multiple clocks, in OOP-style VHDL

```
if (not rst_n) then
    ReuseParallel_0.reset(self);
else
    if rising_edge(clk0) then
        MAC_0.update_registers(self.mac0); -- update 'mac0' on 'clk0' rising edge
    end if;

    if rising_edge(clk1) then
        MAC_1.update_registers(self.mac1); -- update 'mac1' on 'clk1' rising edge
    end if;
end if;
```

Synthesis result (Fig. 3.5) show that registers are clocked by different clocks. The reset signal is common for the whole design.

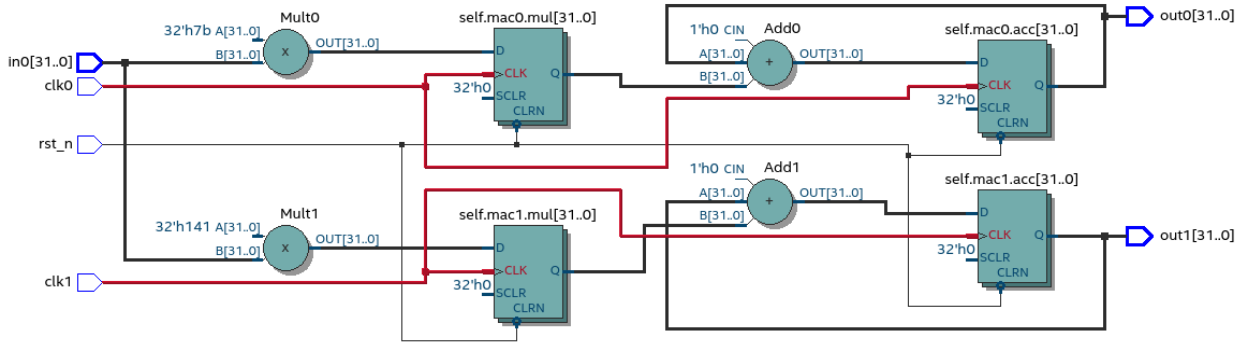

Fig. 3.5: Synthesis result with modified top-level process (Intel Quartus RTL viewer)

3.2 Converting Python to VHDL

The Python to VHDL conversion process relies heavily on the results of last chapter, that allows sequential OOP Python code easily map to VHDL. Even so, converting Python syntax to VHDL poses some problems.

The biggest challenge in conversion from Python to VHDL is types, namely Python does not have them, while VHDL has. Conversion process must find all the types for Python variables, the process of this is described in `pyvhdl_types`.

After the types are all known, the design can be converted from Python to VHDL syntax. This requires some way of traversing the Python source code and applying VHDL rated transforms.

Conversion progress requires no understanding of the source code nor big modifications. .. _pyvhdl_types:

3.2.1 Finding the types

Python is dynamically typed language, meaning that types come into play only when the code is running. On the other hand VHDL is statically typed, all the types must be written in soruce code.

The advantage of the Python way is that it is easier to program, no need to define variables and ponder about the types. Downsides are that there may be unexpected bugs when some variable changes type. In some cases dynamic typing may also reduce code readability.

In sense of conversion, dynamic typing poses a major problem, somehow the missing type info should be recovered for the VHDL code. Most straightforward way to solve this is to try finding the variables value from code, for example `a = 5`, clearly type of `a` is integer. Problem with this method is that is much more complex than it initially appears. For example `a = b`. To find the type of 'a' converter would need to lookup type of 'b', these kind of sutffs can get really complex.

Alternative, and what Pyha is using, is to run the Python code so all the variables get some value, the value can be inspected programmically and type inferred. For example, consider the class on Listing 3.11.

Listing 3.11: Example Python class, what are the types?

```
class SimpleClass(HW):
    def __init__(self, coef):
        self.coef = coef

    def main(self, a):
        local_var = a
```

Listing 3.12 show example for getting the type of class variable. It initializes the class with argument `5`, that is assigned to the 'coef' variable. Then `type()` can be used to query the variable type. On the example result is `int`, so this can be converted to VHDL `integer` type.

Listing 3.12: Using `type()` to get type name

```
>>> dut = SimpleClass(5)
>>> dut.coef
5
>>> type(dut.coef)
<class 'int'>
```

Pyha deduces registers initial values in same way, only the first assigned value is considered.

Local variables, like `local_var` and argument `a` on Listing 3.12 are harder to deduce as Python provides no way of accessing function locals scope. Note that locals exsist only in the stack, thus after the function call they are lost forever. Luckly this problem has been

encountered before in *[6]*, which 'hacks' the Python profiling interface in order to save the locals for each function. Pyha uses this approach to keep track of the local values.

Listing 3.13: Function locals variable type

```
>>> dut.main.locals # before any call, locals are unknown
{}
>>> dut.main(1) # call function
>>> dut.main.locals # locals can be extracted
{'a': 1, 'local_var': 1}
>>> type(dut.main.locals['local_var'])
<class 'int'>
```

Advantage of this method is low complexity, another perk is that this way could be used to keep track of all the variable values, in future this can enable the automatic conversion from floating point to fixed point. In addition, this way allows the 'lazy' coding, for example where fixed-point gains the bound limit only during the execution of the design.

Downside is that each function in the design must be executed before conversion is possible. Also the conversion result may depend on the data types that are inputed to the functions, but this can also be an advantage.

3.2.2 Syntax conversion

The syntax of Python and VHDL is surprisingly similar. VHDL is just much more verbose, requires types and Python has indention oriented blocks.

Python provides some tools that simplify the traversing of source files, like abstract syntax tree (AST) module and lib2to3. These tools work by parsing the Python file into a tree structure, that can be then traversed and modified. For example the MyHDL conversion is based on this. This method works but is quite complex and requires alot of code.

Lately new project has emerged called RedBaron *[7]*, that aims to simplify operations with Python source code. It features rich tools for searching and modifing the source code. Unlike AST it also keeps all the formatting in the code, including comments. RedBaron parses the source code into rich objects, for example the `a = 5` would result in a `AssignmentNode` object that has an `__str__` function that instruct how these kind of objects are written out.

Pyha overwrites the `__str__` method to instead of `=` print `:=` and also add `;` to the end of statement. Resulting in a VHDL compatible statement `a := 5;`. Beauty of this is that this simple modification actually turns **all** the Python style assignments to VHDL style.

Listing 3.14 shows a more complex Python code that is converted to VHDL (Listing 3.15), by Pyha. Most of the transforms are obtained by the same method described above. Some of the transforms are a bit more complex, like figuring out what variables need to be defined in VHDL code.

Listing 3.14: Python function to be converted to VHDL

```
def main(self, x):
    y = x
    for i in range(4):
        y = y + i

    return y
```

Listing 3.15: Conversion of Listing 3.14 assuming `integer` types

```
procedure main(self:inout self_t; x: integer; ret_0:out integer) is
    variable y: integer;
begin
    y := x;
    for i in 0 to (4) - 1 loop
        y := y + i;
    end loop;

    ret_0 := y;
end procedure;
```

3.2.3 Comparison to other methods

Like HLS must do much work to deduce registers.. Pyha can convert basically line by line, very simple.

Todo

??

3.3 Summary

The sequential object-oriented VHDL model is one of the contributions of this thesis. It has been developed to provide simpler conversion from Python to VHDL. Pyha converts directly to the VHDL model by using RedBaron based syntax conversions. Type information is reuqired trough the simulation before conversion.

Chapter 4

Summary

This work introduced new Python based HDL language called Pyha. That is an sequential object-oriented HDL language. Overview of Pyha features have been given and shown that the tool is suitable to use with model based DSP systems. It was also demonstrated that Pyha supports fixed point types and semi-automatic onversion from floating point types. Pyha also provides good support for unit test and designs are debuggable.

Two case studies were presented, first the moving average filter that was implemented in RTL level. Second example demonstrated that blocks written in Pyha can be reused in pure Python way, developed linear phase DC removal filter by reusing the implementation of moving average filter. Synthesisability has been demonstrated on Cyclone IV device (BladeRF).

Another contribution of this thesis is the sequential object-oriented VHDL model. This was developed to enable simple conversion of Pyha to VHDL. One of the advantages of this work compared to other tools is the simplicyty of how it works.

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

Future perspectives are to implement more DSP blocks, especially by using GNURadio blocks as models. That may enable developing an work-flow where GNURadio designs can easily be converted to FPGA. In addition, the Pyha system could be improved to add automatic fixed point conversion and support for multiple clock domains.

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] Aurelian Ionel Munteanu. Gotcha: access an out of bounds index for a systemverilog fixed size array. URL: http://www.amiq.com/consulting/2016/01/26/gotcha-access-an-out-of-bounds-index-for-a-systemverilog-fixed-size-array/.

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

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

[4] Jan Decaluwe. Why do we need signal assignments? URL: http://www.jandecaluwe.com/hdldesign/signal-assignments.html.

[5] IEEE. Ieee standard vhdl language reference manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, 2009.

[6] Pietro Berkes and Andrea Maffezoli. Decorator to expose local variables of a function after execution. URL: http://code.activestate.com/recipes/577283-decorator-to-expose-local-variables-of-a-function-/.

[7] Laurent Peuch. Redbaron: bottom-up approach to refactoring in python. URL: http://redbaron.pycqa.org/.

[8] Wim Meeus, Kristof Van Beeck, Toon Goedemé, Jan Meel, and Dirk Stroobandt. An overview of today's high-level synthesis tools. *Design Automation for Embedded Systems*, 16(3):31–51, 2012. URL: http://dx.doi.org/10.1007/s10617-012-9096-8, doi:10.1007/s10617-012-9096-8.

[9] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, 33–36. New York, NY, USA, 2011. ACM. URL: http://doi.acm.org/10.1145/1950413.1950423, doi:10.1145/1950413.1950423.

[10] F. Winterstein, S. Bayliss, and G. A. Constantinides. High-level synthesis of dynamic data structures: a case study using vivado hls. In *2013 International Conference on Field-Programmable Technology (FPT)*, 362–365. Dec 2013. doi:10.1109/FPT.2013.6718388.

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

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

[13] Hayden Kwok-Hay So. *BORPH: An Operating System for FPGA-Based Reconfigurable Computers*. PhD thesis, UNIVERSITY OF CALIFORNIA, BERKELEY, 2007.

[14] Stuart Sutherland and Don Mills. Synthesizing systemverilog busting the myth that systemverilog is only for verification. *SNUG Silicon Valley*, 2013.

[15] Ieee p1076 working group vhdl analysis and standardization group (vasg). URL: http://www.eda-twiki.org/cgi-bin/view.cgi/P1076/WebHome.

[16] Open source vhdl verification methodology (osvvm). URL: http://osvvm.org/.

[17] Lars Asplund. Vunit. URL: http://vunit.github.io/.

[18] Robert Ghilduta and Brian Padalino. Bladerf vhdl ads-b decoder. URL: https://www.nuand.com/blog/bladerf-vhdl-ads-b-decoder/.

[19] Neil Lawrence. Gpy: moving from matlab to python. URL: http://inverseprobability.com/2013/11/25/gpy-moving-from-matlab-to-python.

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

[21] Amulya Vishwanath. Enabling high-performance floating-point designs. URL: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01267-fpgas-enable-high-performance-floating-point.pdf.

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

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

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