TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Thomas Johann Seebeck Department of Electronics

Gaspar Karm, IVEM153410

PYHA - OBJECT-ORIENTED HARDWARE DESCRIPTION LANGUAGE BASED ON PYTHON

Master's Thesis

Supervisors:
Muhammad Mahtab Alam
PhD

Yannick Le Moullec
PhD

Tallinn 2017

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Thomas Johann Seebecki elektroonikainstituut

Gaspar Karm, IVEM153410

PYHA - PYTHONIL PÕNINEV OBJEKTORIENTEERITUD RIISTVARAKIRJELDUSKEEL

Magistritöö

Juhendajad:
Muhammad Mahtab Alam
PhD

Yannick Le Moullec
PhD

Tallinn 2017

Author's declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Gaspar Karm
14.05.2017

Abstract

Pyha -
Object-Oriented Hardware Description Language

The task of implementing digital signal processing systems on hardware is often carried out by using high-level tools, such as MATLAB to HDL converter. However, these tools can easily cost over tens of thousands of euros, making them unusable for reproducible research and open source designs. Other high-level tools are mostly based on 'C', which is not suitable for modeling and prototyping purposes. Given the limitations and drawbacks of existing solutions, this thesis proposes Pyha, a new Python based hardware description language aimed at simplifying DSP hardware development in an open-source manner. Pyha raises the abstraction level by suggesting the object-oriented programming model and provides fixed-point type support. In addition, this work makes an effort to simplify the testing process of hardware systems by providing better simulation interface for unit-testing and debugging capabilities in Python domain.

The thesis is in English and contains 39 pages of text, 5 chapters, 24 figures.

Annotatsioon

Pyha -
Pythonil põninev objektorienteeritud riistvarakirjelduskeel

Digitaalse signaalitööstuse süsteemide riistvara loomiseks kasutatakse enamasti kõrgetasemelisi tööriistu, nagu näiteks 'MATLAB HDL Coder'. Need tööriistad pakuvad küll kõrget produktiivsust aga võivad maksta kümneid tuhandeid eurosid, mille tõttu pole need lahendused kasutatavad reprodutseeritava teadustöö ning avatud lähetkoodiga projektide jaoks. Enamus alternatiivseid lahendusi põhineb 'C' keelel, mis ei sobi modellerimiseks ega prototüüpimiseks. Vastavalt alternatiivse lahenduste puudustele, pakub see lõputöö välja Python programmeerimiskeelel põhineva riistvarakirjelduskeele, nimega Pyha, mille eesmärk on lihtsustada digitaalsete süsteemide arendamist. Lahendus põhineb avatud lähtekoodil. Pyha põnineb objektorienteeritud programmeerimisel ning toetab püsikomaarve. Lisaks on lihtsustatud riistvara testimimine, tänu lihtsustatud simulatsioonide jooksutamise liidesele, võimalik on Pythoni domeenis programmi silumine.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 39 leheküljel, 5 peatükki, 24 joonist.

Contents

List of Figures

Chapter 1

Introduction

The most commonly used tools today for designing digital hardware are VHDL and SV (SystemVerilog). SV is aggressively promoted by the big EDA (electronic design automation) companies (Cadence, Mentor, Synopsys) along with UVM (Universal Verification Methodology). In 2003, Aart de Geus, Synopsys CEO, stated that SV will replace VHDL in 10 years *[1]*. It is true that tool vendors have stopped improving VHDL support, but advancements are being made in the open source community e.g. VHDL-2017 standard *[2]*. In addition, active development is done for the open source simulator GHDL *[3]*, Open Source VHDL Verification Methodology (OSVVM) *[4]* and unit-testing library VUnit *[5]*. All of these advancements aim to ease the verification aspects while the synthesis part has mostly stayed the same for the past 10 years.

Numerous projects exist that propose to use higher level HDL (hardware description language), in order to raise the abstraction level. For example, MyHDL turns Python into a hardware description and verification language *[6]*, or CλaSH *[7]*, purely Haskell based functional language developed at University of Twente. Recently Chisel *[8]* has been gaining some popularity, it is an hardware construction language developed at UC Berkeley that uses Scala programming language, providing functional and object-oriented features.

On the other front, HLS (high-level synthesis) tools aim to automate the refinement from the algorithmic level to RTL (register-transfer level) *[9]*. Lately the Vivado HLS, developed by Xilinx, has been gaining popularity. As of 2015, it is included in the free design suite of Vivado (device limited). The problem with HLS tools is that they are often promoted as direct C to RTL tools but in reality often manual code transformations and guidelines are needed, in order to archive reasonable performance *[10]*.

The DSP (digital signal processing) systems can be described in previously mentioned HLS or

HDL languages, but the most productive way is to use MATLAB/Simulink/HDLConverter flow, which allows users to describe their designs in Simulink or MATLAB and use HDL convertible blocks provided by MATLAB or FPGA tool vendor *[11]*.

1.1 Problem statement

There is no doubt that MATLAB based workflow offers a highly productive path from DSP models to hardware. However, these tools can easily cost over tens of thousands of euros and often FPGA vendor tools are required, which adds additional annual cost *[11]*. Using these tools is not suitable for reproducible research and is completely unusable for open source designs. Thus, the designers must turn to alternative design flows; for example *[12]* provides a hardware implementation of an ADS-B (automatic dependent surveillance – broadcast) receiver. First, they did the prototyping in the MATLAB environment, the working model was then translated to C for real-time testing and fixed-point modeling. Lastly, the C model was manually converted to VHDL.

Given the limitations and drawbacks of existing solutions, this thesis proposes Pyha, a new Python based hardware description language aimed at simplifying DSP hardware development in an open-source[1] manner. Pyha raises the RTL design abstraction level by enabling sequential and object-oriented style. DSP systems can be built by using the fixed-point type and semi-automatic conversion from floating point. In addition, this work makes an effort to simplify the testing process of hardware systems by providing better simulation interface for unit-testing.

The basis of Pyha is Python, a general purpose programming language that is especially well suited for rapid prototyping and modeling. Python has also found its place in scientific projects and academia by offering most of what is familiar from MATLAB, free of charge. Scientists are already shifting from MATLAB to Python in order to conduct research that is reproducible and accessible by everyone *[13]*. Fig. 1.1 shows the popularity comparison (based on Google searches) of Python, MATLAB and C. As can be seen, Python is the only language with positive slope, soon Python will be the most popular programming language in the world (currently Java).

[1] Repository: https://github.com/gasparka/pyha

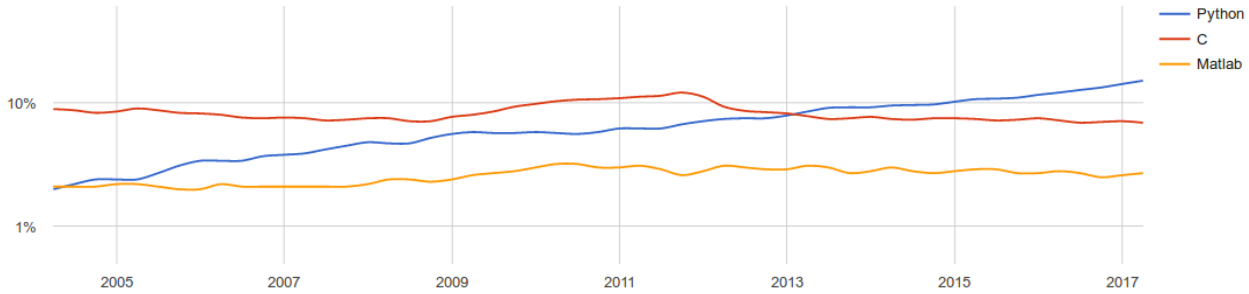

Fig. 1.1: Programming language popularity *[14]*. Python 15.1%, C 6.9%, MATLAB 2.7%

1.2 Structure of the thesis

After gaining the context and problem statement in the current chapter, Section 2 presents the proposed hardware description language Pyha. Next, Section 3 develops the object-oriented VHDL model and deals with the problem of converting Python to VHDL. Section 4 shows how Pyha can be used to implement medium complexity DSP systems and gives a comparison to existing tools. Section 5 concludes this thesis and suggest ideas for future work. The related work is introduced and discussed throughout this thesis, thus no specific literature review chapter has been included.

Chapter 2

Hardware design with Pyha

This chapter introduces the main contribution of this thesis, Pyha - a tool to design digital hardware in Python. The fist half of the chapter demonstrates how basic hardware constructs can be defined, using Pyha. Follows the introduction to fixed-point type.

Note: The examples in the first half of this chapter are based on `integer` types in order to reduce complexity.

2.1 Introduction

Conventional HDL languages promote concurrent and entity oriented models which can be confusing for software developers. In this thesis, Pyha has been designed as a sequential object-oriented language, that works directly on Python code. Using a sequential design flow is much easier to understand and is equally well synthesizable as shown in this thesis. Object-oriented design helps to better abstract the RTL details and ease design reuse.

For illustration purposes, Listing 2.1 shows an example Pyha design. The `main` function has been chosen as a top level entry point, other functions can be used as pleased.

Listing 2.1: Simple combinatory design, implemented in Pyha

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

```
        b = a * 314

        if a == 9:
            b = 0
        return a, b
```

One of the contributions of this thesis is a sequential OOP VHDL model which is used to simplify conversion from Pyha to VHDL. The example of the VHDL conversion is shown in Listing 2.2. The OOP VHDL model is developed and examined in Section 3.

Listing 2.2: The `main` funciton of Listing 2.1 converted to OOP VHDL

```
procedure main(self:inout self_t; x: integer; ret_0:out integer; ret_
↪1:out integer) is
    variable a: integer;
    variable b: integer;
begin
    a := x + 1 + 3;
    b := a * 314;

    if a = 9 then
        b := 0;
    end if;

    ret_0 := a;
    ret_1 := b;
end procedure;
```

Fig. 2.1 shows the synthesis result. The `a` output is formed by adding '1' and '3' to the `x` input. Next the `a` signal is compared to `9`; if equal, `b` is outputted as 0, otherwise `b = a * 314`. This complies exactly with the Python and VHDL descriptions.

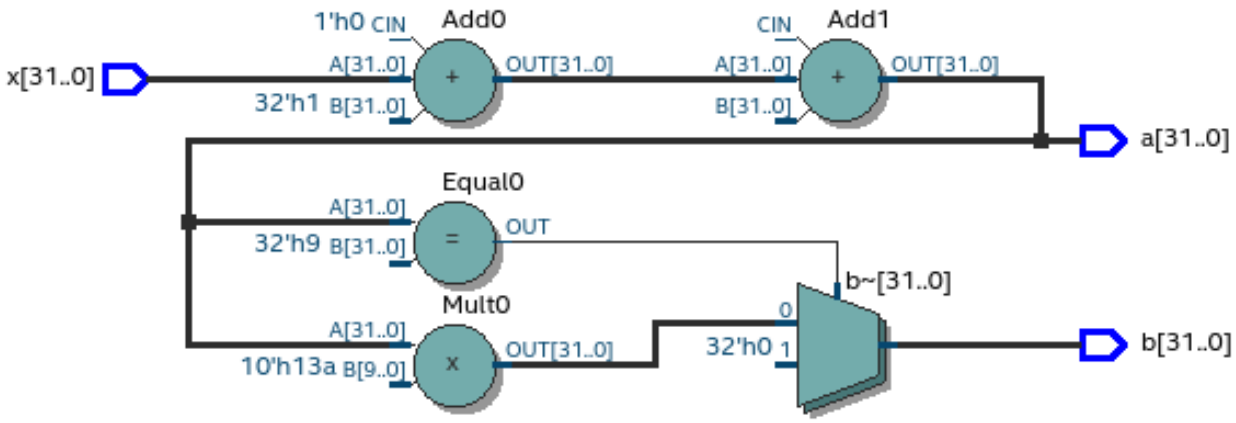

Fig. 2.1: Synthesised RTL of Listing 2.2 (Intel Quartus RTL viewer)

One aspect of hardware design that Pyha aims to improve is testing. Conventional testing flow requires the construction of testbenches that can be executed using simulators.

Pyha has been designed so that the synthesis output is behaviourally equivalent to the Python run output, this means that Pyha designs can use all the Python debugging tools. Fig. 2.2 shows a debugging session on the Listing 2.1 code, this can drastically help the development process.

```
def main(self, x):  self: <main.Basic object at 0x7f2c21f7a630>  x: 5
    a = x + 1 + 3  a: 9
    b = a * 314  b: 2826
    if a == 9:
        b = 0
    return a, b
```

Fig. 2.2: Debugging using PyCharm (Python editor)

Furthermore, unit testing is accelerated by providing `simulate(dut, x)` function, that runs the following simulations without any boilerplate code:

- Model: this can be any Python code that fits as an high level model;
- Pyha: like Listing 2.1, Python domain simulation;
- RTL: simulation in VHDL domain, Pyha model is converted to VHDL;
- GATE: synthesises the VHDL code, using Intel Quartus, and simulates the resulting gate-level netlist.

This kind of testing function enables test-driven development, where tests can be first defined for the model and fully reused for later RTL implementation. Listing 2.3 shows an example unit test

for the `Basic()` module. Python `assert` statements can be used for unit test development. Pyha also provides `assert_simulate(dut, expected, x)` function that automatically compares the output list to the `expected` list.

Listing 2.3: Unit test for the Basic module

```
x = [1, 2, 3, 4, 5, 6, 7, 8]
dut = Basic()
y = simulation(dut, x) # y contains result of all simulations
# assert something
```

2.2 Sequential logic

The way how registers are inferred is a fundamental difference between the HDL and HLS languages. HDL languages leave the task to the designer, while HLS languages automate the process. In this work, Pyha has been designed to follow the HDL language approach, because this simplifies the conversion to VHDL. Extensions can be considered in future editions.

In conventional programming, state is usually captured by using class variables which can retain values between function calls. Inspired from this, all the class variables in Pyha are handled as registers.

2.2.1 Accumulator example

Consider the design of an accumulator (Listing 2.4); it operates by sequentially adding up all the input values of every successive function call.

Listing 2.4: Accumulator implemented in Pyha

```
class Acc(HW):
    def __init__(self):
        self.acc = 0

    def main(self, x):
        self.next.acc = self.acc + x
        return self.acc
```

The class structure in Pyha has been designed so that the `__init__` function shall define all the memory elements in the design, the function itself is not converted to VHDL, only the variables are extracted. For example `__init__` function could be used to call `scipy.signal.firwin()` to design FIR filter coefficients, initial assignments to class variables are used for register initial/reset values.

The `self.next.acc = ...`, simulates the hardware behaviour of registers i.e. the delayed assignment. In general, this is equivalent to the VHDL <= operator. Values are transferred from **next** to **current** before the `main` call. In Pyha each call to the `main` function can be considered as an clock edge.

The synthesis results displayed in the Fig. 2.3 shows the adder and register, that is the expected result for accumulator.

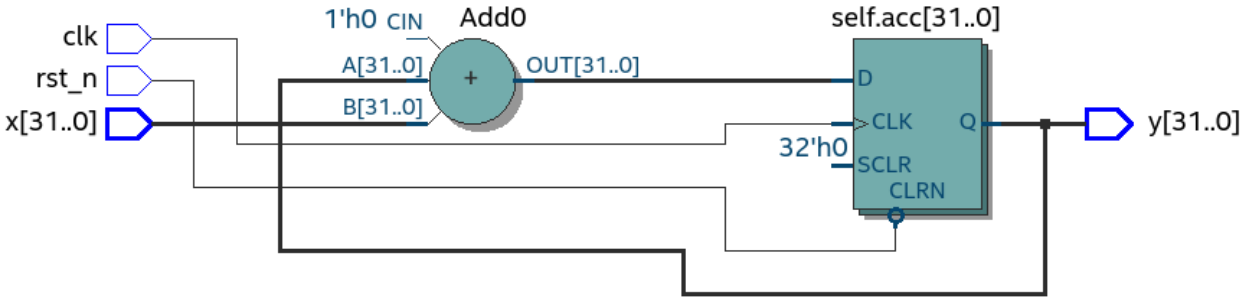

Fig. 2.3: Synthesis result of Listing 2.4 (Intel Quartus RTL viewer)

One inconvenience is that every register on the signal path delays the output signal by 1 sample, this is also called pipeline delay or latency. The delay can be seen from Fig. 2.4, where hardware related simulations are delayed by 1 sample as compared to the software model.

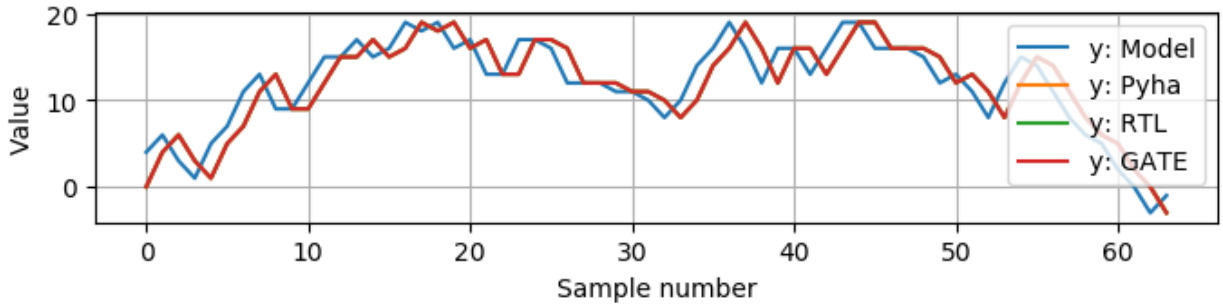

Fig. 2.4: Simulation of the `Acc` module, input is a random integer [-5;5]. Hardware simulations are delayed by 1, caused by the register

Pyha reserves a `self._delay` variable, that hardware classes can use to specify their delay.

Simulation functions read this variable to compensate the simulation outputs. Setting the `self._delay = 1` in the `__init__` function would shift the hardware simulations left by 1 sample, so that all the simulation would be exactly equal. This functionality is useful for documenting the delay of modules and simplifies the development of unit-tests.

2.2.2 Block processing and sliding adder

A common technique required to implement DSP systems is block processing, i.e. calculating results on a block of input samples. Until now, the `main` function has worked with a single input sample, registers can be used to keep history of samples, so that block processing can be applied.

For example, consider an algorithm that outputs the sum of last 4 input values. Listing 2.5 shows the Pyha implementation, it works by keeping history of 4 last input samples and summing them for output.

Listing 2.5: Sliding adder algorithm, implemented in Pyha

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

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

        # add all elements in 'shr'
        sum = 0
        for x in self.shr:
            sum = sum + x

        # register the output
        self.next.y = sum
        return self.y
```

The `self.next.shr = [x] + self.shr[:-1]` implements an '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 designs. Fig. 2.5 shows the synthesis results.

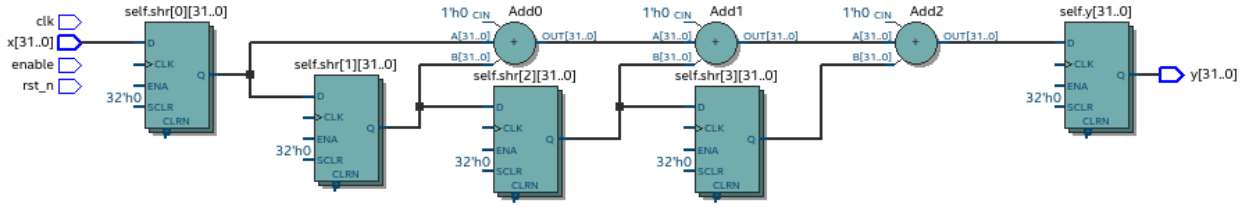

Fig. 2.5: Synthesis result of Listing 2.5 (Intel Quartus RTL viewer)

This design can be made generic by changing the `__init__` function to take the window length as a parameter (Listing 2.6), so that `SlidingAdder(window_len=4)` would add 4 last elements, while `SlidingAdder(window_len=6)` would add 6.

Listing 2.6: Generic sliding adder, `window_len` controls the `shr` list length

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

This design has a few issues when the `window_len` is increased (Fig. 2.6). First, every stage requires a separate adder which increases the resource cost, this also forms a long critical path which in turn decreases the maximum clock rate of the design.

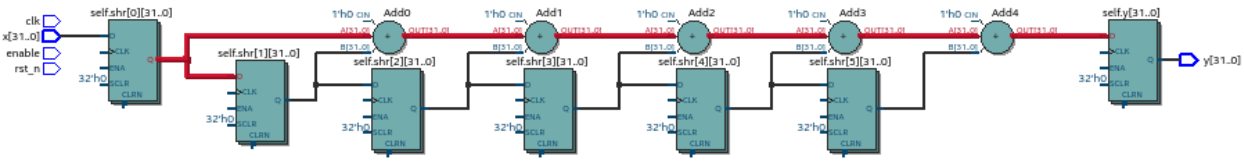

Fig. 2.6: Synthesis result of `SlidingAdder(window_len=6)`, the red line shows the critical path (Intel Quartus RTL viewer)

Conveniently, the algorithm can be optimized to use only 2 adders, no matter the window length. Listing 2.7 shows that instead of summing all the elements, the overlapping part of the previous calculation can be used to significantly optimize the algorithm.

Listing 2.7: 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]

# optimized way to calculate by reusing previous results (recursive)
y[5] = y[4] + x[10] - x[4]
y[6] = y[5] + x[11] - x[5]
```

Listing 2.8 gives the implementation of the optimal sliding adder; it features a new register `sum`, that keeps track of the previous output.

Listing 2.8: Optimal sliding adder, implemented in Pyha

```
class OptimalSlideAdd(HW):
    def __init__(self, window_len):
        self.shr = [0] * window_len
        self.sum = 0 # register to remember the 'last' sum

        self._delay = 1

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

        # add new 'x' to sample and subtract the delayed 'x'
        self.next.sum = self.sum + x - self.shr[-1]
        return self.sum
```

Fig. 2.7 shows the synthesis result. Now the critical path is 2 adders, no matter the `window_len`. In addition, notice how the `shr` is just a stack of registers to delay the input signal.

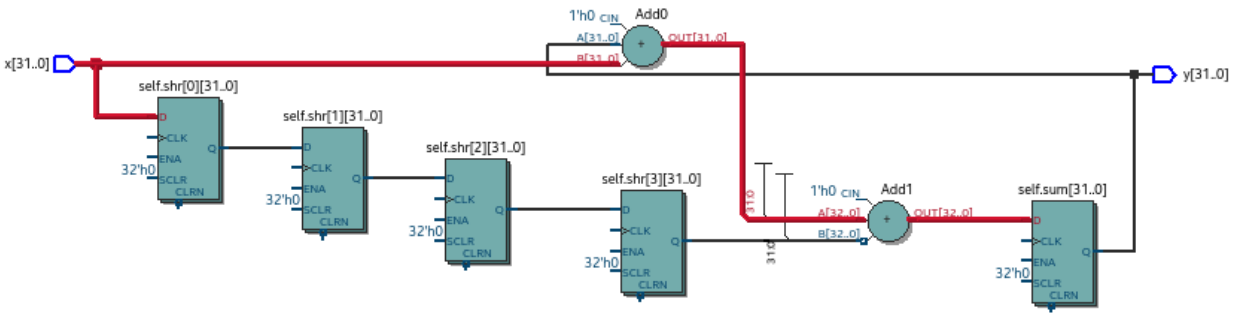

Fig. 2.7: Synthesis result of `OptimalSlideAdd(window_len=4)` (Intel Quartus RTL viewer)

2.3 Fixed-point designs

DSP systems are commonly described in floating-point arithmetic, which are supported by all conventional programming languages. Floating-point arithmetic can also be used in RTL languages, but the problem is high resource usage *[15]*. The alternative is to use fixed-point numbers, that work with integer arithmetic. Another benefit of fixed-point numbers is that they can map to FPGA DSP blocks, thus providing higher clocks speed and reduced resource use[1].

The common workflow is to experiment and write model using the floating-point arithmetic, then convert to fixed-point for hardware implementation. One contribution of this thesis is the implementation of fixed-point class for the Python domain.

2.3.1 Fixed-point support in Pyha

In this work, Pyha has been designed to support signed fixed-point type by providing the `Sfix` class. The implementation maps directly to the VHDL fixed-point library *[17]*[2], that is already known in the VHDL community and proven to be well synthesizable.

`Sfix` class works by allocating bits to the `left` and `right` side of the decimal point. Bits to the `left` determine the integer bounds (sign bit is implicit), while the `right` bits determine the minimum resolution of the number. For example, `Sfix(left=0, right=-17)` represents a number between [-1;1] with resolution of 0.000007629 (`2^-17`). Listing 2.9 shows a few examples on how reducing the `right` reduces the number precision.

[1] Some high-end FPGAs also include floating-point DSP blocks *[16]*

[2] https://github.com/FPHDL/fphdl.

Listing 2.9: Example of `Sfix` type, more bits give better accuracy

```
>>> 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 and recommended fixed-point type in Pyha has been chosen to be `Sfix(left=0, right=-17)`, because it can represent normalized numbers and fits into FPGA DSP blocks *[18] [15]*. Keeping block inputs and outputs in the normalized range can simplify the overall design process. More details about the fixed-point implementation can be found in Section A.

2.3.2 Converting sliding adder to fixed-point

Consider converting the sliding window adder (developed in Section 2.2.2) to a fixed-point implementation. This requires changes only in the `__init__` function (Listing 2.10).

Listing 2.10: Fixed-point sliding adder, the rest of the code is identical to the one in Section 2.2.2

```
def __init__(self, window_size):
    self.shr = [Sfix()] * window_size # lazy type
    self.sum = Sfix(left=0)           # always resize left to 0
```

The first line sets `self.shr` to store `Sfix()` elements, this is a lazy statement as it does not specify the fixed-point bounds i.e. it will take bounds from the first assignment to the `self.shr` variable. The `Sfix(left=0)` forces `left` to 0 bits, while the fractional part is determined by the first assign. One problem with the VHDL fixed-point library is that the designer is constantly forced to resize the value to desired format, this thesis has automated this step i.e. every assign to fixed-point variable is resized to the initial format, the bounds may be taken from the assigned value if initial value is lazy.

Synthesis results in Fig. 2.8 show that inputs and outputs are now 18-bits wide, this is due the use of default fixed-point type. Another main addition is the saturation logic, which prevents the wraparound behaviour by saturating the value instead. Wraparound related bugs can be very hard to find, thus it is suggested to keep saturation logic enabled when the overflows are possible.

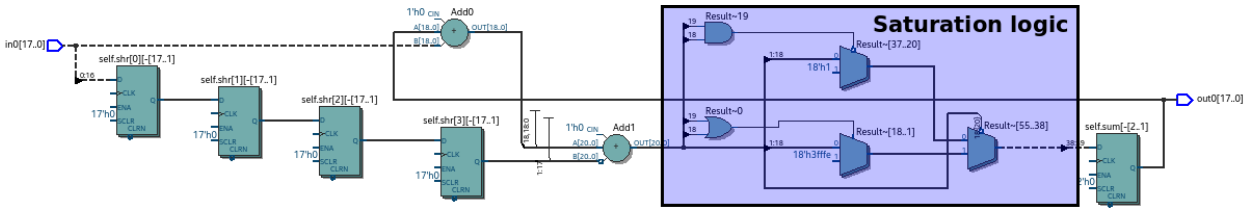

Fig. 2.8: RTL of fixed-point sliding adder, default fixed-point type (Intel Quartus RTL viewer)

The `simulate` function in Pyha has been designed to automatically convert floating-point inputs to fixed-point, the same goes for outputs. This way the unit-test can be kept simple, Listing 2.11 gives an example.

Listing 2.11: Pyha enables testing of fixed-point design with floating-point numbers

```
dut = OptimalSlidingAddFix(window_len=4)
x = np.random.uniform(-0.5, 0.5, 64) # random signal in [-0.5, 0.5]␣
↪range
y = simulate(dut, x)                 # all outputs are floats
# assert or plot results
```

The simulation results shown in Fig. 2.9, show that the hardware related simulations differ from the model. This is because the model is implemented in floating-point arithmetic while hardware typing is limited to [-1;1] range. Notice that the mismatch starts when the value rises over `1.0`.

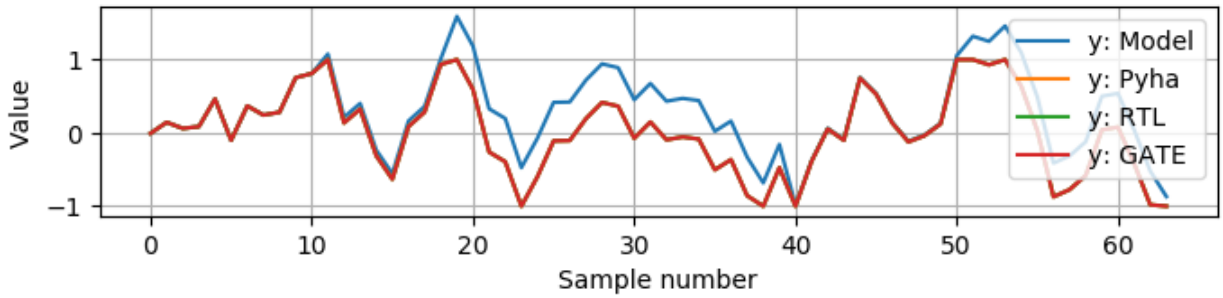

Fig. 2.9: Simulation results of fixed-point sliding sum, input is random signal in [-0.5; 0.5] range

2.4 Summary

This chapter has demonstrated the major features of the proposed tool and the motivation behind them. It was shown that Pyha is an sequential object-oriented programming language based on

Python. It falls in the category of behavioral languages, meaning that the output of Python program is equivalent to the output of the generated hardware. Pyha provides `simulate` functions to automatically and without any boilerplate code run model and hardware related simulations, this helps the design of unit-tests. In addition, Pyha designs are fully debuggable in Python ecosystem. Class variables are used to define registers, this has been inspired by traditional programming languages. 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 accelerates the design process.

Chapter 3

Synthesis

The majority of the hardware related tools end up converting to either VHDL or SystemVerilog, because these are supported by the synthesis tools. Generally the higher level language is converted to very low level VHDL/ SV code, resulting in a complex conversion process and unreadable code for a human.

This thesis tests an alternative path by contributing the sequential object-oriented (OOP) programming model for VHDL. The main motivation is to use it as a conversion target for higher level languages. Major advantages are that the conversion process is simple and the output VHDL is readable and structured.

VHDL has been chosen over 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 *[19]*. In the future both could be supported.

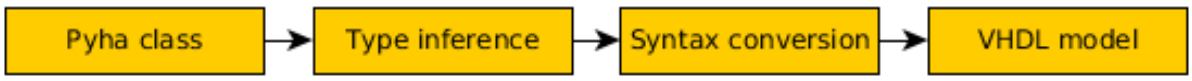

Fig. 3.1: Process of converting Pyha to VHDL.

Fig. 3.1 shows the process of converting Pyha to VHDL. This relies heavily on the final VHDL model (Section 3.1). The process of type inference and syntax conversion is described in Section 3.2.

3.1 Structured, Object-oriented style for VHDL

Structured programming in VHDL has been studied by Jiri Gaisler in *[20]*. He showed that combinatory logic is easily described by functions with sequential statements and proposed the 'two-process' design method, where the first process is used to describe combinatory logic and the second process describes registers. This thesis contributes to the 'two process' model by adding an object-oriented approach, which allows fully sequential designs, easier reuse and removes the one clock domain limitation.

The basic idea behind OOP is to strictly define functions that can perform actions on some group of data. This idea fits well with hardware design, as 'data' can be thought of as registers, and combinatory logic as functions that perform operations on the data. VHDL has no direct support for OOP but it can still be used by grouping data in record (same as C struct) and passing it as parameter to functions. This is essentially the same way how C programmers do it.

Listing 3.1 demonstrates pipelined multiply-accumulate (MAC), written in OOP VHDL. Recall that all the items in the `self_t` record are expected to synthesise as registers.

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; -- return via 'out' argument
end procedure;
```

The synthesis results (Fig. 3.2) 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 that is not placed on the critical path.

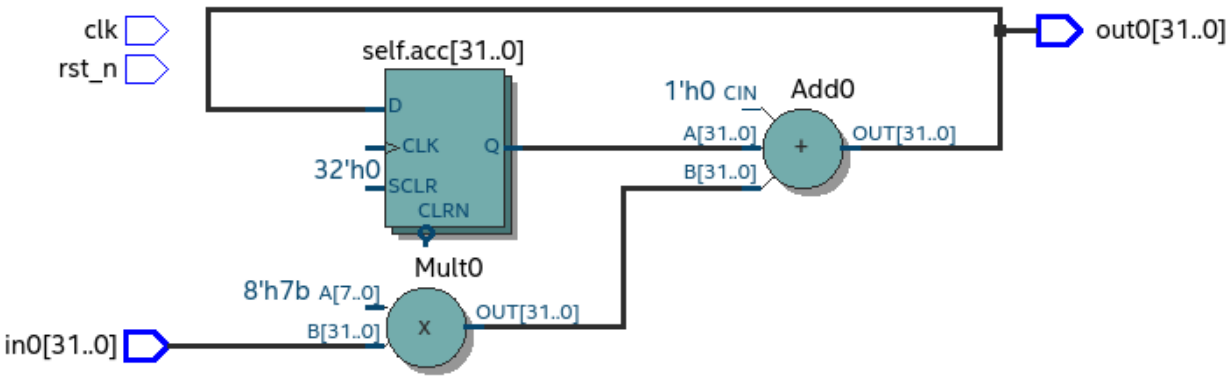

Fig. 3.2: Unexpected synthesis result of Listing 3.1, `self.coef=123` (Intel Quartus RTL viewer)

3.1.1 Defining registers with variables

The previous section made a mistake of expecting the registers to work exactly in the same way as 'class variables' in traditional programming languages. In reality the difference is that registers have delayed assignment i.e. they take value on next clock edge (or in this case, function call).

VHDL defines a special 'signal assignment' operator for this kind of delayed assignment, that can be used on VHDL signal objects like `a <= b`. These objects are hard to map to higher level languages and have limited usage in VHDL structured code constructs.

Conveniently, the signal assignment can be mimicked with two variables, to represent the **next** and **current** values. The signal assignment operator sets the value of **next** variable. On the next simulation delta, all the signals are updated i.e. **next** written to **current**. This way of writing sequential logic has been previously suggested by Pong P. Chu in his VHDL book *[21]* and is also used in MyHDL signal objects *[22]*.

Adapting this style for the MAC example is shown in Listing 3.2; the data model now includes the `nexts` member, that should be used to write register values.

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

```
type next_t is record -- new record to hold 'next' values
    mul: integer;
    acc: integer;
    coef: integer;
end record;

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

    nexts: next_t; -- new element
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;
```

VHDL signal assignment automatically updates the signal values, this has to be done manually when using the two variable method. Listing 3.3 defines the 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 a combinatory function.

Synthesising the revised code shows that the pipelined MAC has been implemented (Fig. 3.3). The `coef` variable is not synthesised to register as it is determined to be constant.

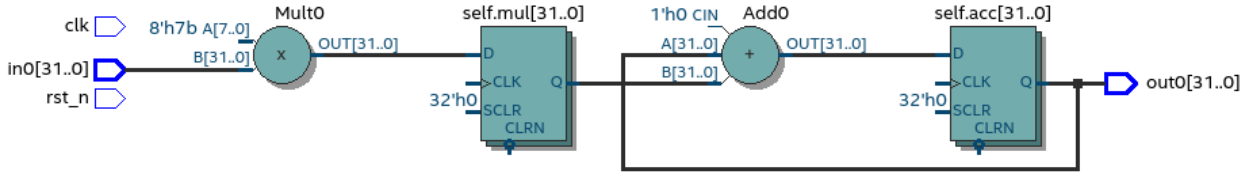

Fig. 3.3: Synthesis result of the revised code (Intel Quartus RTL viewer)

3.1.2 The OOP model

The OOP model, developed in this thesis, consists of the following elements:

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

VHDL supports 'packages', that can be used to group all these elements into common namespace. Listing 3.4 shows the template package for VHDL 'class'. All the class functionality is now in

common namespace.

Listing 3.4: Class template for OOP style VHDL

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

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

    -- function prototypes
end package;

package body MAC 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;
```

The 'reset' function sets the initial values for registers. Listing 3.5 shows a reset function for the MAC circuit. Note that the hardcoded `self.nexts.coef := 123;` could be replaced with VHDL package generics.

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;
```

The hardcoded `self.nexts.coef := 123;` could be replaced with package generic, for example `coef`. Then each new package could define a 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);
```

3.1.3 Use cases

This section demonstrates how instances of VHDL 'classes' can be applied for design reused. Consider an example that consists of two MAC instances and aims to connect them in series (Listing 3.7). In `main`,

Listing 3.7: 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);        -- MAC_0 -> MAC_1
    MAC_1.main(self.mac1, out_tmp, ret_0=>ret_0);    -- MAC_1 -> output
end procedure;
```

The synthesis result shows that two MACs are connected in series, see Fig. 3.4.

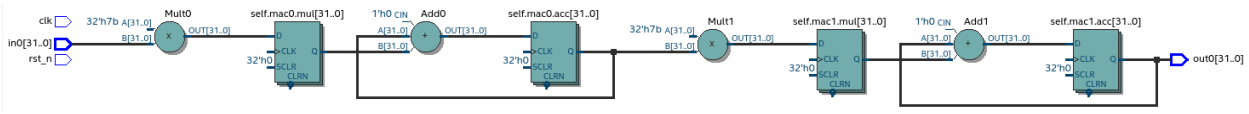

Fig. 3.4: Synthesis result of Listing 3.7 (Intel Quartus RTL viewer)

Connecting two MACs instead in parallel can be done with a simple modification to `main` function to return both outputs (Listing 3.8).

Listing 3.8: 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. 3.5.

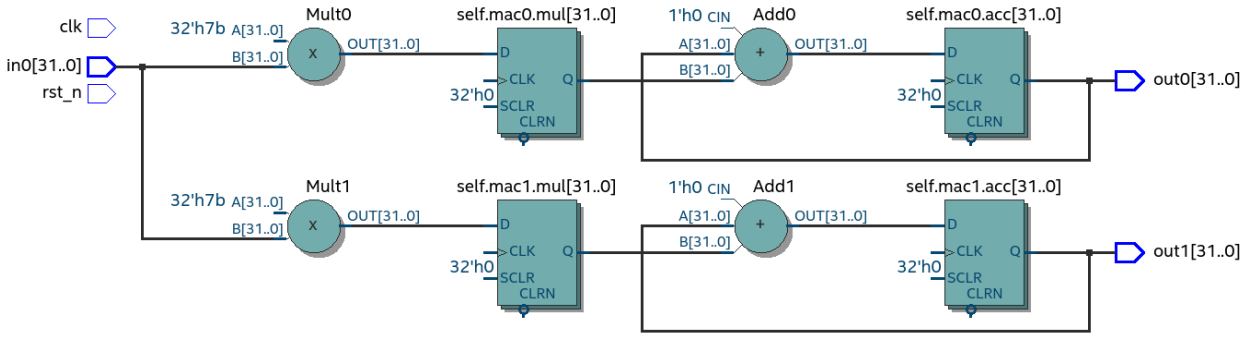

Fig. 3.5: Synthesis result of Listing 3.8 (Intel Quartus RTL viewer)

Multiple clock domains can be easily supported by updating registers at different clock edges. By reusing the parallel MACs example, consider that MAC_0 and MAC_1 are specified to work in different clock domains. For this, only the top level process must be modified (Listing 3.9), the rest of the code remains the same.

Listing 3.9: Top-level for multiple clocks, in OOP-style VHDL

```
if (not rst_n) then
    ReuseParallel_0.reset(self); -- reset everything
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 results (Fig. 3.6) show that registers are clocked by different clocks. The reset signal is common for the whole design.

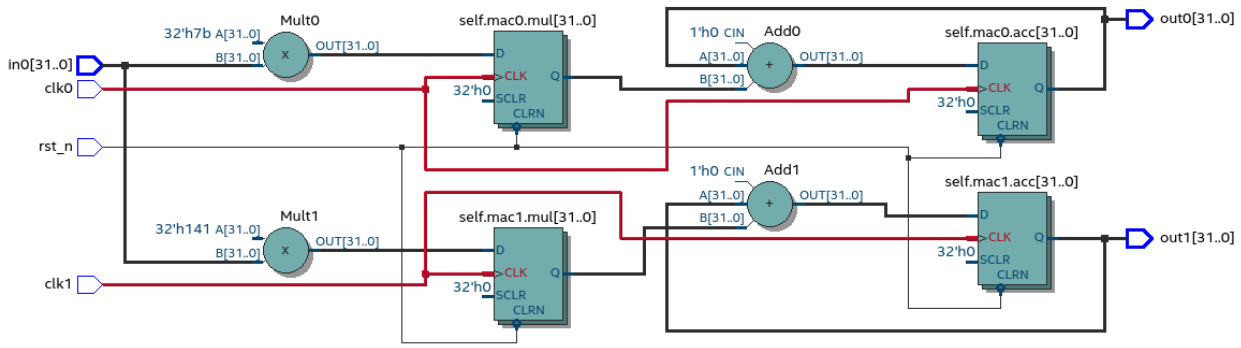

Fig. 3.6: Synthesis result with modified top-level process, MACs work in different clock domains (Intel Quartus RTL viewer)

3.2 Converting Python to VHDL

Tools like MATLAB feature an complex conversion process that requires ‘understanding’ of the source code. Tools must perform major transformations to turn source code into VHDL. This thesis proposes to use simple conversion from Python to VHDL, that requires no majord transformations; this is made possible by the OOP VHDL model that allows easy mapping of Python constructs to

VHDL. Even so, the conversion process poses some challenges like type inference and syntax conversion, that are investigated in this chapter.

3.2.1 Type inference

One of the significant differences between Python and VHDL is the typing system. Python uses dynamic typing i.e. types are determined during code execution, while VHDL is statically typed. This poses a major problem for conversion, as the missing type information in Python sources must be somehow inferred in order to produce VHDL code. A naive way to tackle this problem is to try inferring the types directly from code, for example clearly the type of '`a = 5`' is integer. However, typically the task is more complex; consider Listing 3.10 as an example, no types can be inferred from this code.

Listing 3.10: What are the types of `self.coef`, `a` and `local_var`?

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

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

An alternative is to follow the definition of dynamic typing and execute the code, after what the value can be inspected and type inferred. Listing 3.11 shows this method applied on the class variable, the Python function `type()` can be used to query the variable type.

Listing 3.11: Solving the problem for class variables

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

This solves the problem for class values. Unfortunately, this method is not applicable for the local variables of functions, because these only exist in the stack. This problem has been encountered before in *[23]*, which proposes to modify the Python profiling interface in order to keep track of function local variables. It has been decided to apply this method in Pyha; an example is shown in Listing 3.12.

Listing 3.12: Solving the problem for local variables

```
>>> dut.main.locals # locals are unknown before call
{}
>>> 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'>
```

In sum, this method requires the execution of the Python code before types can be inferred. The main advantage of this is very low complexity. In addition, this allows the usage of 'lazy' fixed point types as shown in Section 2.3. This method can also be used to keep track of all the values a variable takes, which enables automatic conversion from floating-point to fixed-point. The code execution needed for conversion is automated in the `simulate` functions by running the Python domain simulation.

3.2.2 Syntax conversion

Python provides tools that simplify the traversing of source files, like abstract syntax tree (AST) module, that works by parsing the Python file into a tree structure, which can then be modified. Using AST for syntax conversion is known to work but it has very low abstraction level, thus most of the time resulting in complex conversion process. RedBaron *[24]* is a recent high-level AST tool, that aims to simplify operations with Python source code; unlike AST, it also keeps the code formatting and comments.

RedBaron parses the source code into rich objects, for example '`a = 5`' would result in an `AssignmentNode`. Nodes can be overwritten to change some part of the behaviour. For example, the `AssignmentNode` can be modified to change = to `:=` and add `;` to the end of statement. Resulting in a VHDL compatible statement '`a := 5;`'. This simple modification turns **all** the assignments in the code to VHDL style assignments.

Listing 3.13 shows a more complex Python code that is converted to VHDL (Listing 3.14), by Pyha. Most of the transforms are obtained by the same method described above. Some of the transforms are a bit more complex, like return statement to output argument conversion.

Listing 3.13: Python function to be converted to VHDL

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

    return y
```

Listing 3.14: Conversion of Listing 3.13, 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;
```

3.3 Summary

This chapter has shown that Pyha achieves synthesizability by converting the Python code to VHDL. 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, enabling almost direct conversion from Python the VHDL by using RedBaron based syntax transformations. Type information is acquired trough the simulation. Pyha provides `simulate` functions that automate the simulation and conversion parts.

The conversion process is one of the advantages of Pyha, compared to other similar tools. The process has low complexity and produces well formatted and readable VHDL. In addition, syntax conversion could be easily extended to support other conversion targets like SystemVerilog or C.

Chapter 4

Case studies

This chapter demonstrates that Pyha is already usable for real designs. First, an moving average filter is designed, that is later reused for the linear-phase DC removal filter. Pyha has also been used in bigger designs such as frequency-shift keying demodulator, this work is not included in this thesis due due to the time constrains. Last section of this chapter provides an comparison of Pyha to other related tools.

4.1 Moving average filter

The moving average (MA) is the easiest digital filter to understand and use. It is optimal filter for reducing random noise while retaining a sharp step response *[25]*. In communication systems, MA is widely used as an matched filter for rectangular pulses. Fig. 4.1 shows an example of applying MA filter to reduce noise on harmonic signal. Higher window length (averaged over more elements) reduces more noise but also increases the complexity and delay of the filter (MA is a special case of FIR filter *[25]*).

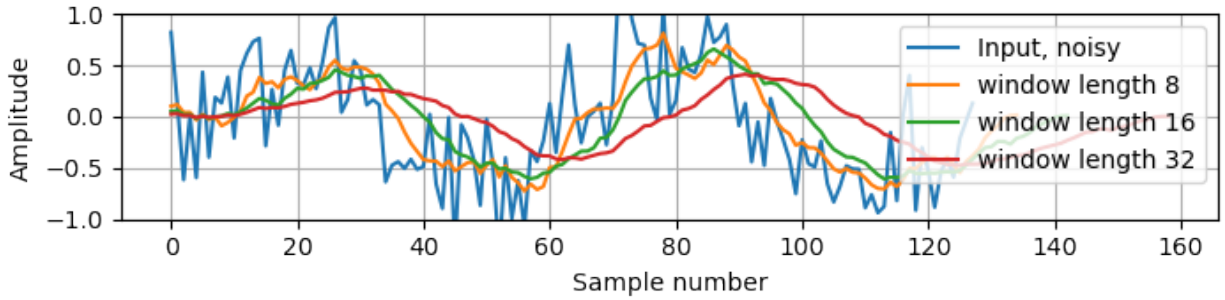

Fig. 4.1: Moving average filter applied on noisy signal, coded in Listing 4.1

Good noise reduction performance can be explained by the frequency response of the MA filter (Fig. 4.2), showing that it is a low-pass filter. The passband width and stopband attenuation are controlled by the window length.

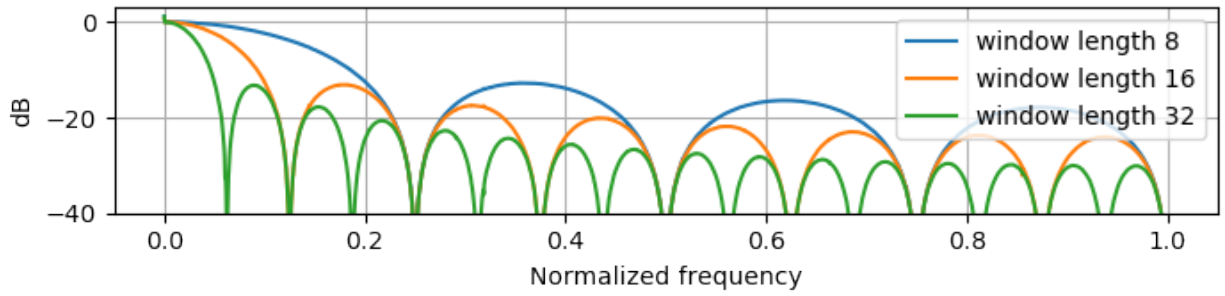

Fig. 4.2: Frequency response of the moving average filter, coded in Listing 4.1

The MA filter is implemented by sliding sum, that is divided by the sliding window length. The division can be carried out by a shift operation if divisor is a 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`. This guarantees that the `sum` is always in the [-1;1] range and no saturation logic is needed.

Listing 4.1 shows the MA filter implementation in Pyha. It is based on the sliding sum, that was implemented in Section 2.3.2. Minor modifications are commented in the code.

Listing 4.1: MA implementation in Pyha

```
class MovingAverage(HW):
    def __init__(self, window_len):
        # calculate power of 2 value of 'window_len', used for division
        self.window_pow = Const(int(np.log2(window_len)))

        # 'overflow_style' turns the saturation off
        self.sum = Sfix(0, 0, -17, overflow_style=fixed_wrap)
        self.shr = [Sfix()] * window_len
        self._delay = 1

    def main(self, x):
        # divide by shifting
        div = x >> self.window_pow

        self.next.shr = [div] + self.shr[:-1]
```

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

Fig. 4.3 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.

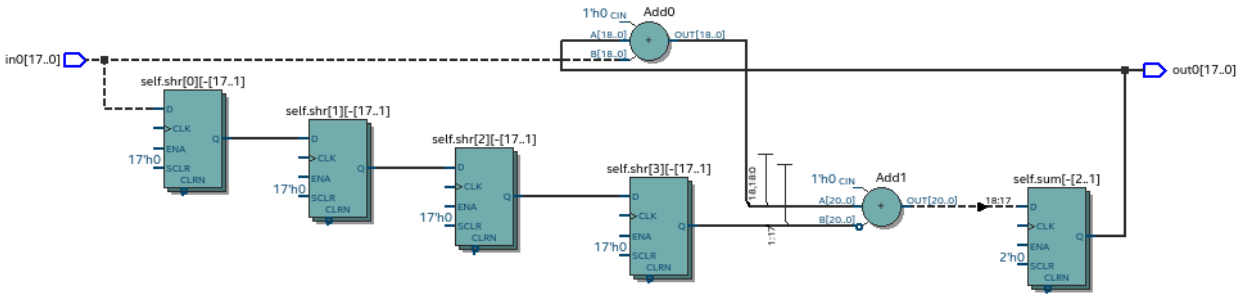

Fig. 4.3: RTL view of moving average (Intel Quartus RTL viewer)

Fig. 4.4 shows simulation results of MA filter used for matched filtering. The plot in (a) shows digital input signal that is corrupted by noise. Plot (b) shows that the MA with a window length equal to the number of samples per symbol can recover (optimal result) the signal from the noise. Next the signal could be sampled to recover bit values (0.5=1, -0.5=0).

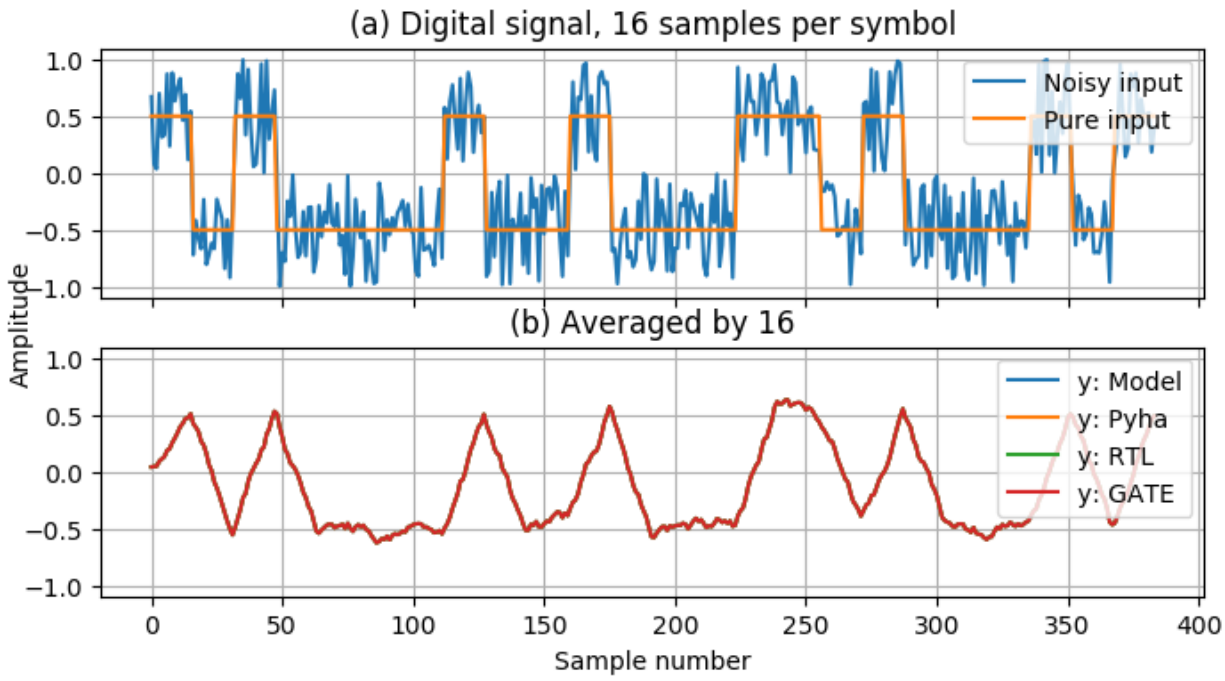

Fig. 4.4: Moving average as matched filter. (b) noisy input signal, (a) averaged by 16, Pyha simulations

4.2 Linear-phase DC removal Filter

This section demonstrates how the object-oriented nature of Pyha can be used for simple design reuse by chaining multiple MA filters to implement 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 *[26]*. DC offset looks like a peak near the 0 Hz on the frequency response. In time domain it manifests as a constant component on the harmonic signal.

In *[27]*, 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. 4.5) *[27]*.

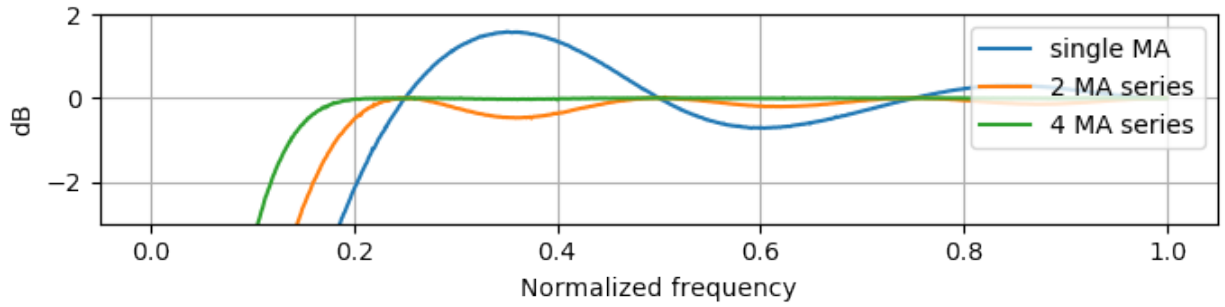

Fig. 4.5: Frequency response of DC removal filter (MA window length is 8)

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 4.2 shows the Pyha implementation.

Listing 4.2: Linear-phase DC removal filter, implemented in Pyha

```
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
        dc = x
        for mav in self.mavg:
            dc = mav.main(dc)

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

This implementation is not exactly following that of *[27]*. They suggest to delay-match the step 1 and 2 of the algorithm, but since the DC component is more or less stable, this can be omitted.

Fig. 4.6 shows that the synthesis generated 4 MA filters that are connected in series, output of the chain is subtracted from the input.

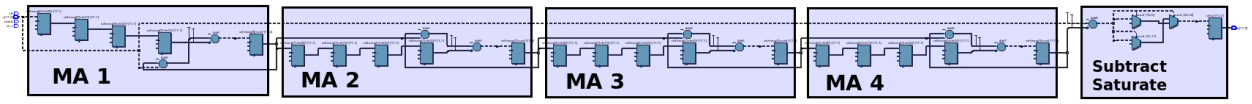

Fig. 4.6: Synthesis result of `DCRemoval(window_len=4)` (Intel Quartus RTL viewer)

In a real application, one would want to use this component with a larger `window_len`. Here 4 was chosen to keep the synthesis result simple. For example, using `window_len=64` gives much better cutoff frequency (Fig. 4.7); FIR filter with the same performance would require hundreds of taps *[27]*.

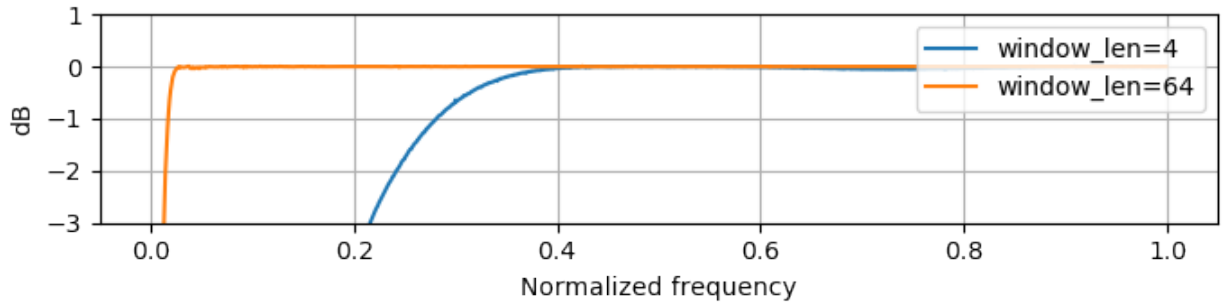

Fig. 4.7: Comparison of frequency response, it depends on `window_len` parameter

This implementation is also very light on the FPGA resource usage (Listing 4.3).

Listing 4.3: Cyclone IV FPGA resource usage for `DCRemoval(window_len=64)`, (Intel Quartus synthesis report)

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

Fig. 4.8 shows the simulation results for input signal with DC component of +0.5, the output of the filter starts countering the DC component until it is removed.

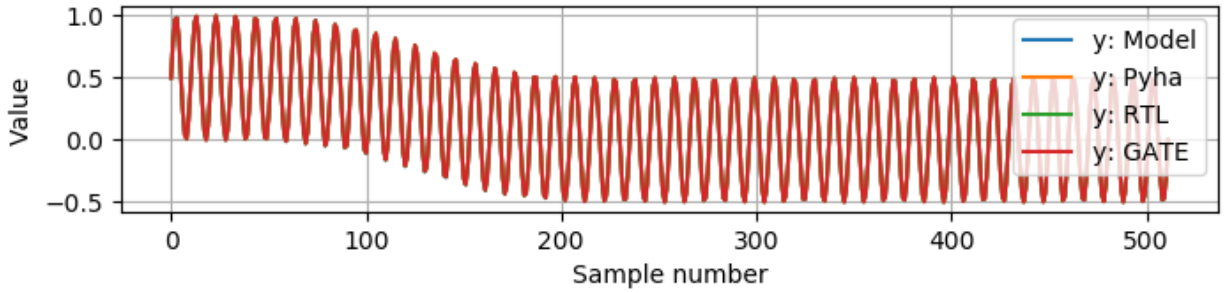

Fig. 4.8: Simulation of DC-removal filter in the time domain, all the simulations are considered equal

4.3 Comparison to similar tools

Traditional HDL languages like VHDL and SV work on large number of concurrent statements and processes that are connected with signals. This is known as event-based style, when some signal changes it may trigger the execution of processes. The reasoning behind this model is that it models exactly how the hardware works. However, the major downside is implementation and readability complexity. The sequentially executed programming style, proposed in this thesis, is much more familiar for software programmers and, as shown in this thesis, results in the same hardware outcome. This work also raises the abstraction level by opening up the Python ecosystem for hardware developers. In addition, the simulations functions provided by Pyha greatly increase the testing productivity and enable test-driven development.

MyHDL is a hardware description language that is also based on Python, but works in the same event-driven way as VHDL/SV. The convertible subset of MyHDL is limited to function based designs, this work proposes object-oriented design method, that is much easier to understand for software developers and eases the design reuse. In general the synthesizable subset of MyHDL

is limited, it has been found that the tool is more useful for high-level modeling purposes *[28]*. MyHDL also does not implement fixed-point type support, thus it is not oriented on DSP designs.

The MATLAB based DSP to HDL tools work on similar abstraction levels as Pyha i.e. code execution is sequential, but user input is required on the placement of registers. Pyha support object-oriented designs while MATLAB is function based like MyHDL. Working with registers and reusing the design is simpler in Pyha. The Simulink flow is based mostly on connecting together already existing blocks. As shown in this chapter, Pyha blocks can be connected easily and in purely Pythonic way. MATLAB also offers an floating-point to fixed-point conversion tool (for additional 10000$ *[29]*). Pyha matches this with semi-automatic conversion by supporting lazy vector bounds, the conversion process is suitable for future implementation of fully automatic conversion.

The C based high level synthesis tools try to turn the behaviour model directly to the RTL level i.e. they automatically infer the register placements and concurrency. However, there are studies that suggest that the productivity gain of these tools is equivalent to the advanced HDL languages like MyHDL or Chisel *[30] [31]*. This is because more often the C algorithm must be modified (and annotated) to suite the hardware *[32] [10]*. On the other hand there are also studies that find the HLS tools to be the only way forward *[33]*, the truth is probably somewhere between. These tools ara mainly gaining popularity, because they appeal to designers coming from software development. This is also the case for Pyha, as it uses pure Python classes and functions. In general the Python based flow provides much higher abstraction than C, also Python is better suited for modeling purposes.

Chapter 5

Conclusion

The task of implementing DSP systems on hardware is often carried out by using high level tools, such as MATLAB. However, these tools are costly, thus not available for everyone and unsuitable for open-source designs. Other high-level tools are mostly based on 'C', which is not reasonable for modeling purposes. Given the limitations and drawbacks of existing solutions, this thesis has proposed Pyha, a new Python based hardware description language aimed at simplifying DSP hardware development in an open-source manner.

Overview of main features of Pyha have been given and shown that the proposed tool is usable for describing hardware components. It was also demonstrated that Pyha supports fixed-point types and semi-automatic conversion from floating point types. Pyha also provides good support for unit test and enables debugging of hardware designs in Python ecosystem.

Two use cases presented in Section 4 show that the developed solution is already usable on solving real life problems. First, the moving average filter was implemented and demonstrated as a matched filter. The second example showed how the object-oriented nature of Pyha can be used for simple design reuse by developing linear phase DC removal filter.

The comparison to other similar tools (Section 4.3) show that Pyha is an good alternative for commercial tools and provides increased abstraction level compared to other open source tools. Pyha may appeal to designers coming from software programming as it uses regular Python constructs and executes in sequential manner.

5.1 Contributions

The contributions of this thesis are:

- Hardware description, simulation and debugging in Python - this is the main contribution of this thesis;
- Sequential object-oriented VHDL model - the object-oriented VHDL model was developed to allow simple conversion from Python to VHDL;
- Method for converting Python to X - this thesis developed an simple way to convert Python syntax to VHDL, this method could be used for other purposes as well;
- Fixed-point arithmetic library for Python - fixed-point library was developed to support cycle-accurate simulation with the converted VHDL code, this library can be used to model fixed-point systems in Python domain;
- Simplified simulation functions - functions that can execute multiple layers of simulations (Python, RTL, GATE) without any boilerplate code, this contribution significantly improves the testability of hardware designs.

5.2 Future work

The technical part of Pyha has been developed by the author of this thesis during the period of one year; while the work is already usable, it could be definitely improved. For example, finishing the support of automatic conversion from floating-point to fixed-point. The current scope of the Python simulator has been limited to single clock domain, which is suitable for most DSP systems; lifting this limitation could make Pyha acceptable for a wider community.

One of the most interesting enhancements would be the extension to the conversion process, to support some HLS backend (such as VivadoHLS). This would present the designer a choice between describing the RTL with a VHDL backend or higher-level abstractions with an HLS backend.

Long term work is to implement more DSP blocks in Pyha, so that complex systems could be built faster. In addition, the tool should also be supported on Windows based systems.

Appendix A

Fixed-point type details

This appendix provides more information about the fixed-point and complex fixed-point types. Basics have been covered in Section 2.3. The `Sfix` type was built in such a way that all the functions map to the VHDL library *[17]*, so that the conversion process is simple. This has also allowed to verify the implementation against a known working solution.

Practical fixed-point variables can store only a part of what floating-point value could. Converting a design from floating-point to fixed-point, opens up a possibility of overflows i.e. the value grows bigger or smaller than the format can represent. By default Pyha uses fixed-point numbers that have saturation enabled, meaning that the fixed-point bounds are forced when value goes out of bounds. Listing A.1 shows some examples of the saturation process.

Listing A.1: Example of saturation, note that Pyha emits `warning` when the saturation happens

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

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

>>> Sfix(2.5, left=2, right=-17)
2.5 [2:-17]
```

For some designs the wrap-around behaviour of fixed-point variables could be a feature, for example an free running counter, that wraps to 0 when maximum value is reached. For these cases,

saturation can be disabled as shown on Listing A.2.

Listing A.2: Saturation disabled, values wrap around

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

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

Fig. A.1 shows an comparison of the overflow modes. The input is an sine wave that exceeds the fixed-point bounds. In general the saturation logic can minimize the damage.

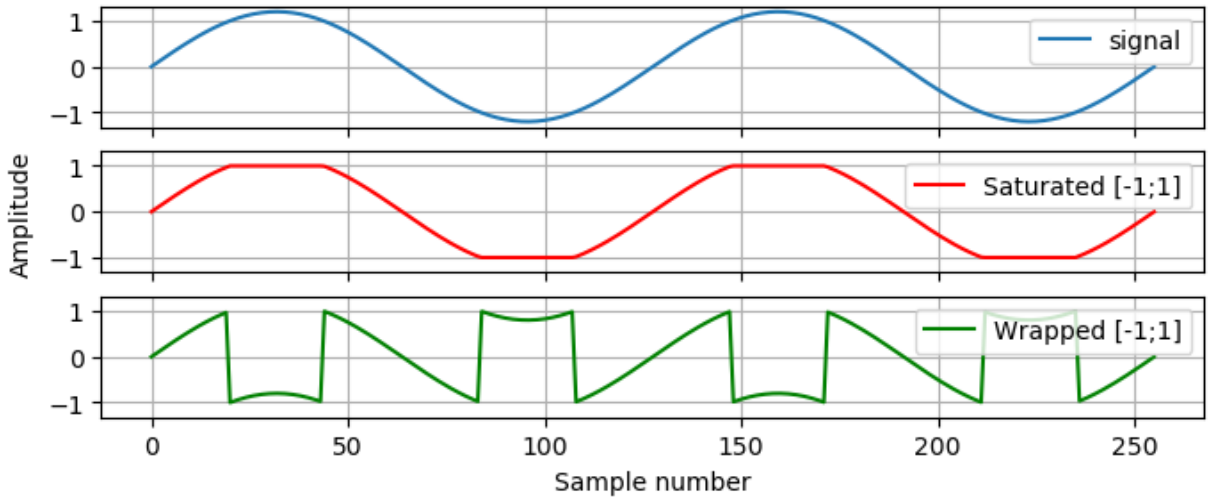

Fig. A.1: Comparison of overflow modes.

Arithmetic operations on fixed-point variables work as usual with an exception of division, that is not defined as it is almost always unnecessary in hardware. The philosophy of fixed point library is to guarantee no precision loss happens during arithmetic operations, in order to do this it has to extend the output format. It is designers job to resize numbers back into optimal format after operations.

Listing A.3 shows an example, where two numbers of range [-1, 1] are added. Library avoids the overflow condition by adding one bit to the integer side, thus guaranteeing that overflows are impossible.

Listing A.3: Arithmetic rules of fixed-point numbers

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

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

Fixed-point number can be forced to whatever size by using the resize functionality, Listing A.4 gives an example of this.

Listing A.4: Arithmetic rules of fixed-point numbers

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

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

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

Pyha support automatic resizing for registers i.e. all assignments to the `self` will be automatically resized to the original type of the initial definition in the `__init__` function.

Objective of this work was to simplify model based design and verification of DSP to FPGA models. One frequent problem with DSP models was that they require the use of complex numbers. In order to unify the interface of the model and hardware model, Pyha supports complex numbers for interfacing means, arithmetic operations are not defined. That means complex values can be passed around and registered but calculations must be done on `.real` and `.imag` elements, that are just Sfix objects. Example of complex fixed-point support is shown on Listing A.5.

Listing A.5: Complex fixed-point type

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

```
>>> a.real
0.4499969482421875 [0:-17]

>>> a.imag
0.8799972534179688 [0:-17]

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

Bibliography

[1] Richard Goering. Designers debate 'VHDL is dead' assertion. URL: http://www.eetimes.com/document.asp?doc_id=1216820.

[2] IEEE P1076 Working Group VHDL Analysis and Standardization Group (VASG). URL: http://www.eda-twiki.org/cgi-bin/view.cgi/P1076/WebHome.

[3] Tristan Gingold. VHDL 2008/93/87 simulator. 2017. URL: https://github.com/tgingold/ghdl.

[4] Open Source VHDL Verification Methodology (OSVVM). URL: http://osvvm.org/.

[5] Lars Asplund. VUnit. URL: http://vunit.github.io/.

[6] MyHDL. URL: http://www.myhdl.org.

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

[8] Jonathan Bachrach. Chisel: Constructing Hardware in a Scala Embedded Language. 2012.

[9] 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.

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

[12] Robert Ghilduta and Brian Padalino. BladeRF VHDL ADS-B decoder. URL: https://www.nuand.com/blog/bladerf-vhdl-ads-b-decoder/.

[13] Neil Lawrence. GPy: Moving from MATLAB to Python. URL: http://inverseprobability.com/2013/11/25/gpy-moving-from-matlab-to-python.

[14] Pierre Carbonnelle. PYPL PopularitY of Programming Language. URL: http://pypl.github.io/PYPL.html.

[15] Ambrose Finnerty and Hervé Ratigner. Reduce Power and Cost by Converting from Floating Point to Fixed Point. 2017.

[16] 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.

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

[18] Altera. Cyclone IV FPGA Device Family Overview. 2016.

[19] 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/.

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

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

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

[23] 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-/.

[24] Laurent Peuch. RedBaron: Bottom-up approach to refactoring in Python. URL: http://redbaron.pycqa.org/.

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

[26] BladeRF community. DC offset and IQ Imbalance Correction. 2017. URL: https://github.com/Nuand/bladeRF/wiki/DC-offset-and-IQ-Imbalance-Correction.

[27] Rick Lyons. Linear-phase DC Removal Filter. 2008. URL: https://www.dsprelated.com/showarticle/58.php.

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

[29] MathWorks' HDL Coder and Verifier: High-Level Synthesis Expands to MATLAB Users. 2012. URL: https://www.bdti.com/InsideDSP/2012/09/05/MathWorks.

[30] O. Arcas-Abella, G. Ndu, N. Sonmez, M. Ghasempour, A. Armejach, J. Navaridas, Wei Song, J. Mawer, A. Cristal, and M. Luján. An empirical evaluation of High-Level Synthesis languages and tools for database acceleration. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, 1–8. Sept 2014. doi:10.1109/FPL.2014.6927484.

[31] Christopher Felton. Little to no benefit from C based HLS. 2015. URL: https://www.fpgarelated.com/showarticle/578.php.

[32] S. Qin and M. Berekovic. A Comparison of High-Level Design Tools for SoC-FPGA on Disparity Map Calculation Example. *ArXiv e-prints*, August 2015. arXiv:1509.00036.

[33] 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.