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 the testing process and DSP systems. In a sense, Pyha can be considered as Python bindings for the object-oriented VHDL style, developed in this thesis.

Main features:

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

1.1 Background

High level synthesis (HLS) languages try to raise the abstraction by enabling writing hardware algortihms in software ways. HLS tools must do complex analysis of input code to match the RTL output, data dependencies and such. In general the HLS world is dominated by C type languages. There have been countless numbers of them both in commercially and academically, for example LegUp is C based tool being developed at the University of Toronto.

In commercial front lately the Vivado HLS have gained some traction, as of Vivado (2015.4) software release it is included in the free downloadable set. It works with C, C++ or SystemC inputs, that can be rather confsing.. In general Xilinx has reportet great adoption for this tools, however there are also people who are sceptical about this. Downside of the HLS

languages is that they use C language that is not expressive for the job, witch starts the need for macro statements to instrcut the synthesis process. Xilinx even suggest in using TCL (long dead scripting language) in cororation with VivadoHLS.

On the other front there are projects that aim to enchance the hardware description languages(HDL). Difference between the HDL and HLS is that the former exactly describe the hardware while the HLS derive the hardware from higher level description. Here are MyHDL, MyHDL is Python to VHDL/Verilog converter, first release dating back to 2003. It turns Python into a hardware description and verification language, providing hardware engineers with the power of the Python ecosystem *[3]*. MyHDL has been used in the design of multiple ASICs and numerious FPGA projects *[2]*.

Recently a new Scala based HDL has been gaining traction, called Chisel. 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 *[8]*. Recently the Chisel has introcuded version 3, that is kind of a rewrite. FIRRTL!

Todo

Clash...or remove?

DSP systems in general fall you yet another product line where MATLAB is the boss, very expensive. In *[10]*, open-sourced a ADS-B decoder, 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.

More common approach is to use MATLAB stack for also the fixed-point simulations and for conversion to VHDL. Also Simulink can be used.

Simulink based design flow has been reported to be used in Berkeley Wireless Research Center (BWRC) *[11]*. Using this design flow, users describe their designs in Simulink using blocks provided by Xilinx System Generator *[11]*.

The problem with such kind of design flow is that it costs alot. Only the MATLAB based parts can easly cost close to 20000 EUR, as the packages depend on eachother. For example for reasonable flow user must buy the Simulink software but that also requires the MATLAB software, in addtion to do DSP, DSP toolbox is needed.. etc. Also the FPGA vendor based tools, like Xilinx System Generator are also expensive and billed annually.

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 *[18]*. Most of the added synthesizable features already existed in VHDL, making the synthesizable subset of these two languages almost equal. In that sense it is highly likely that ideas developed in this chapter could apply for both programming languages.

On the same time the traditional VLSI languages are not standing still -> VHDL OSVM, UNIT TESTS etc..

In the design verification role, SystemVerilog is widely used in the chip-design industry. The three largest EDA vendors (Cadence, Mentor, Synopsys) have incorporated SystemVerilog into their mixed-language HDL-simulators. Although no simulator can yet claim support for the entire SystemVerilog LRM, making testbench interoperability a challenge, efforts to promote cross-vendor compatibility are underway. In 2008, Cadence and Mentor released the Open Verification Methodology, an open-source class-library and usage-framework to facilitate the development of re-usable testbenches and canned verification-IP. Synopsys, which had been the first to publish a SystemVerilog class-library (VMM), subsequently responded by opening its proprietary VMM to the general public. Many third-party providers have announced or already released SystemVerilog verification IP. [WIKIST] In general the big EDA is pushing the SystemVerilog hard, Aart de Geus, Synopsys CEO, has stated that SystemVerilog will replace VHDL `vhdl_dead`. That was in 2003.

Meanwhile the VHDL development is going on mostly in the open-source sphere. Currently there is an VHDL-2017 standard in the works *[19]*. 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) *[20]* and VUnit, that simplifies unit-testing in VHDL.

1.2 Objective

The tool, designed in the process of this thesis, aims to provide an open-source alternative to the mostly MATLAB based DSP flows. 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.

Main design method in Pyha is 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.

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.

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.

It is safe to say that Vivado HLS and others support everything that Pyha does and that makes sense they are devbelopd by big companies and are years ahead. However Pyha has some advantages:

- Trival conversion to VHDL, no magic intended
- Power of full RTL desing, but also abstractable by parametrizable classes (RLT and HLS in one thing)
- Python vs C
- Integration of model based designs to unit process
- Testing simplification, share unit-tests for model and hardware!
- Bridge to VHDL people, build castle and bridge
- This tool actually shows how it works

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.

Sell this!!!. Has good integration for model, is debuggable. Running simulations is extreamly easy. Good fixed point support. Modern software dev tools in hw.

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

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

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

Todo

Comparison to tools already in Python domain? Why do it? SELL Key feature is simplicity?

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.

1.2.1 Using SystemVerilog instead of VHDL

SystemVerilog (SV) is the new standard for Verilog language, it adds significant amount of new features to the language *[18]*. Most of the added synthesizable features already existed in VHDL, making the synthesizable subset of these two languages almost equal. In that sense it is highly likely that ideas developed in this chapter could apply for both programming languages.

Todo

Be careful when using opinions in scientific work. It is fine that you clearly indicate that this is your opinion, but it is maybe safer to rephrase a bit. Or do you have references that also support your opinion?

However, in my opinion, SV is a worse IR language compared to VHDL, because it is much more permissive. For example it allows out-of-bounds array indexing. This 'feature' is actually written into the language reference manual *[21]*. VHDL would error out the simulation, possibly saving debugging time.

While some communities have considered the verbosity and strictness of VHDL to be a downside, in my opinion it has always been an strength, and even more now when the idea is to use it as IR language.

The only motivation for using SystemVerilog over VHDL is tool support. For example Yosys *[22]*, an open-source synthesis tool, supports only Verilog; however, to the best of my knowledge it does not yet support SystemVerilog features. There have been also some efforts in adding a VHDL frontend *[23]*.

Todo

What is the VHDL frontend status?

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!

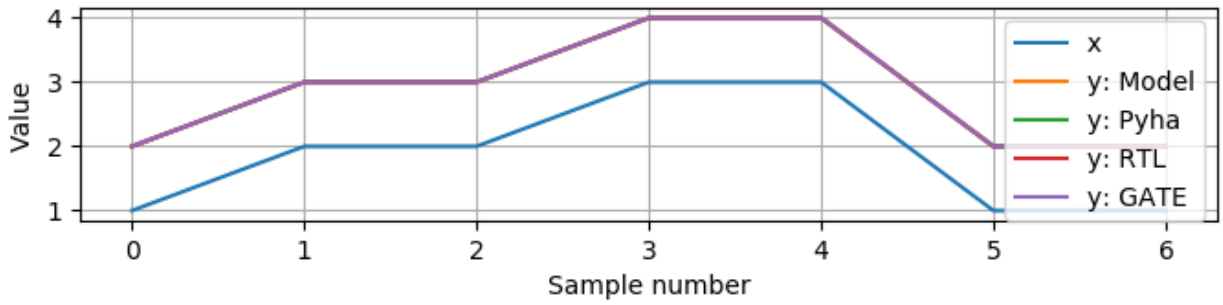

Fig. 2.1: Testing the adder module, `simulation(dut, expect, x)` outputs, all equivalent

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

As an example synthesis target device is 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 *[24]*:

- 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).

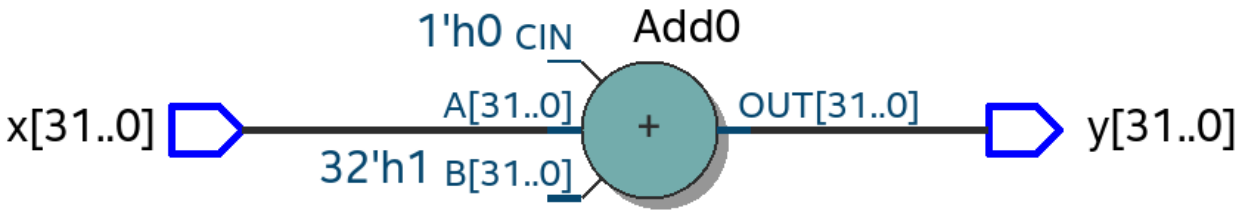

Fig. 2.2: Synthesised RTL of the `Adder(coef=1)` module, `32'1h1` means 32 bit constant with value 1 (Intel Quartus RTL viewer)

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.

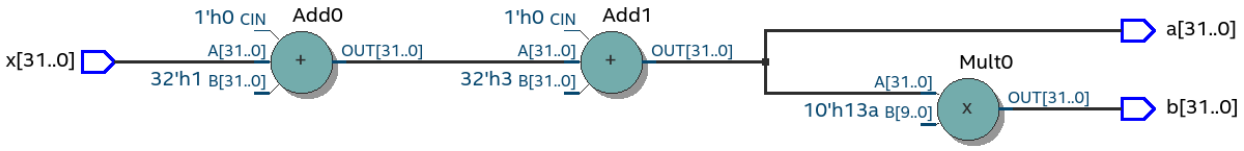

Fig. 2.3: Synthesis result of Listing 2.3 (Intel Quartus RTL viewer)

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!

```
class Basic(HW):
    def main(self, x):  self: <main.Basic object at 0x7f8d30c64390>  x: 1
        a = x + 1 + 3  a: 5
        b = a * 314  b: 1570
        return a, b
```

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.

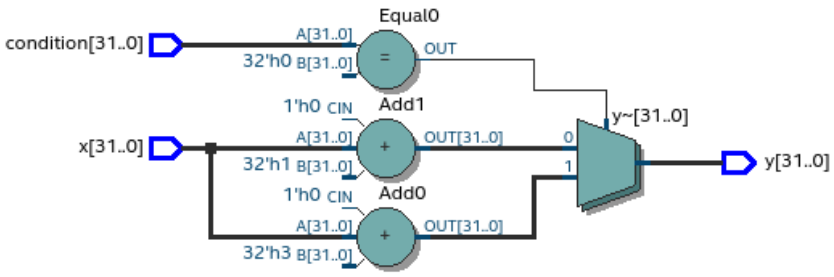

Fig. 2.5: Synthesis result of Listing 2.4 (Intel Quartus RTL viewer)

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.

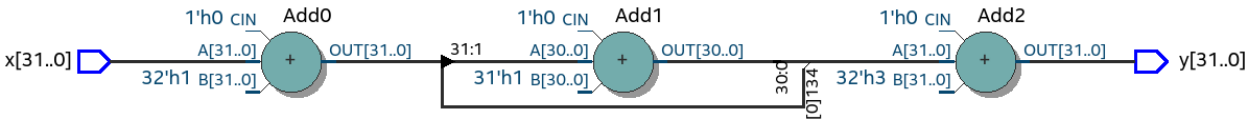

Fig. 2.6: Synthesis result of Listing 2.5 (Intel Quartus RTL viewer)

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.

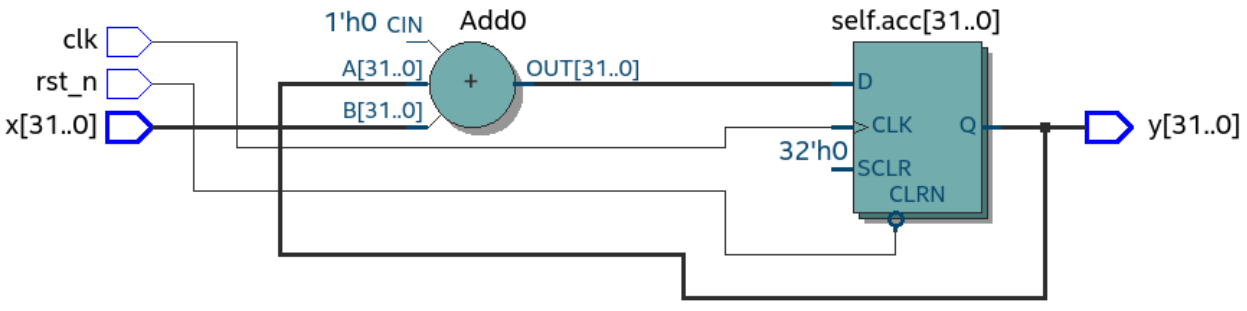

Fig. 2.7: Synthesis result of Listing 2.5 (Intel Quartus RTL viewer)

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.

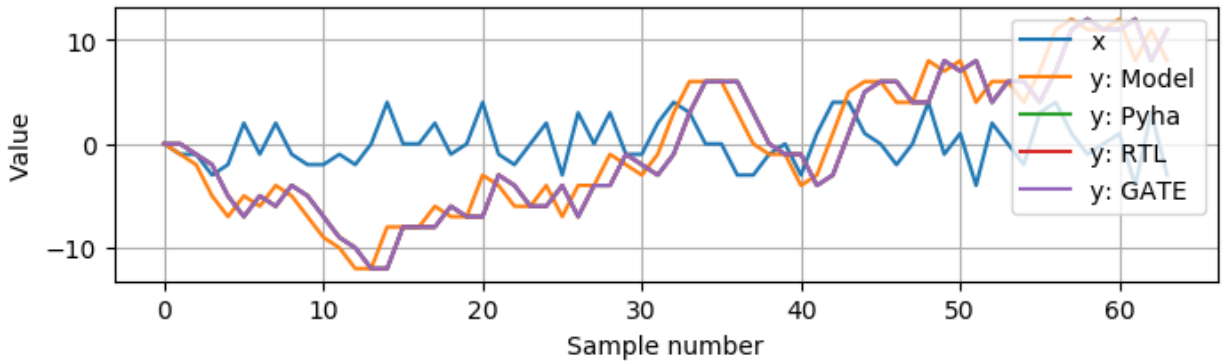

Fig. 2.8: Simulation of the accumulator (x is a random integer [-5;5])

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.

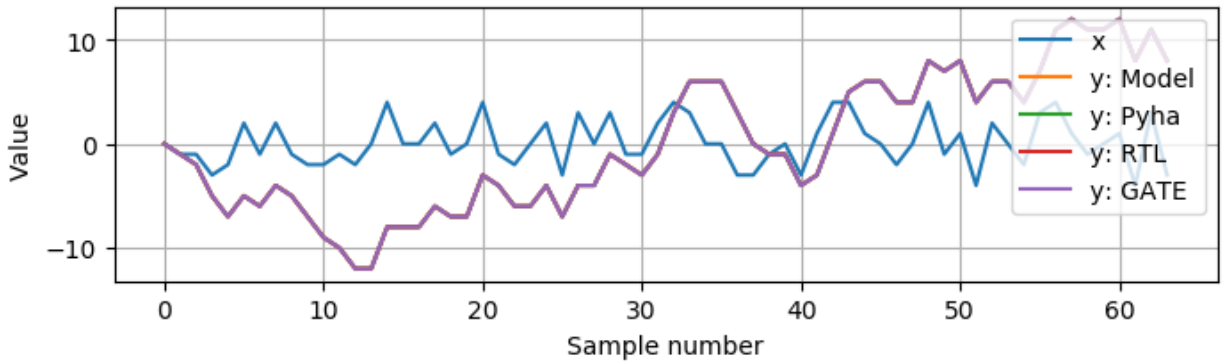

Fig. 2.9: Simulation of the delay-compensated accumulator (x is a random integer [-5;5])

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.

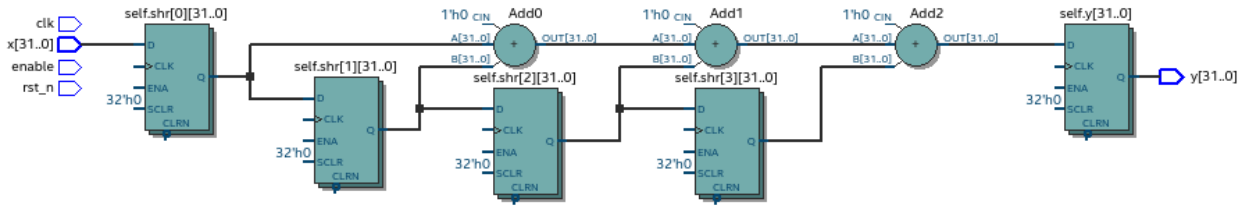

Fig. 2.10: Synthesis result of Listing 2.9 (Intel Quartus RTL viewer)

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?

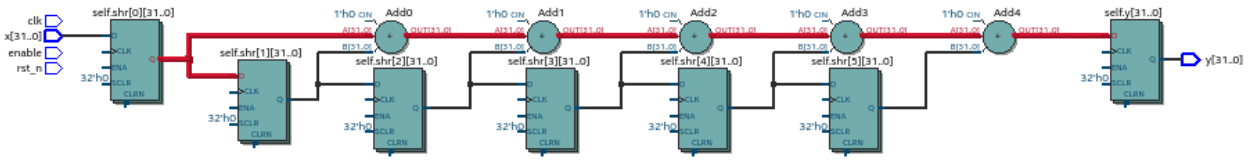

Fig. 2.11: RTL of `window_len=6`, the red line shows the critical path (Intel Quartus RTL viewer)

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.

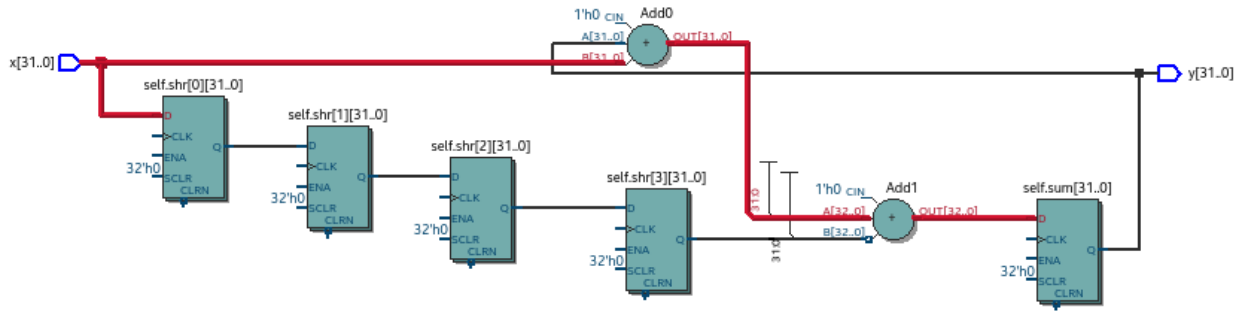

Fig. 2.12: Synthesis result of Listing 2.9, `window_len=4` (Intel Quartus RTL viewer)

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.

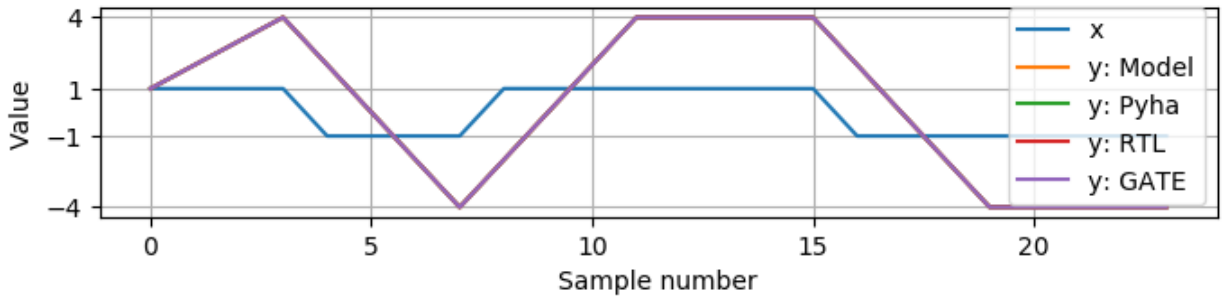

Fig. 2.13: Simulation results for `OptimalSlideAdd(window_len=4)`

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 *[25]*).

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 *[24]*) 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.

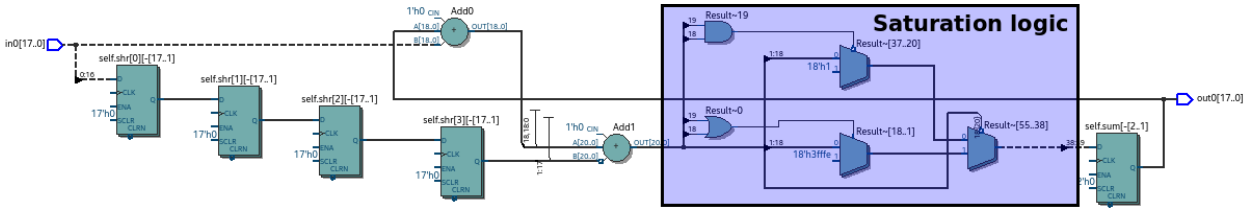

Fig. 2.14: RTL of fixed-point sliding adder (Intel Quartus RTL viewer)

Fig. 2.15 plots the simulation results. 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.

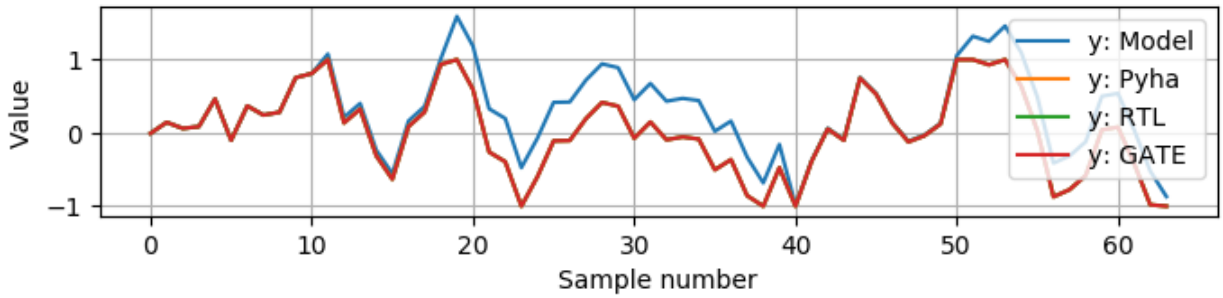

Fig. 2.15: Simulation results of FP sliding sum

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 *[26]*.

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).

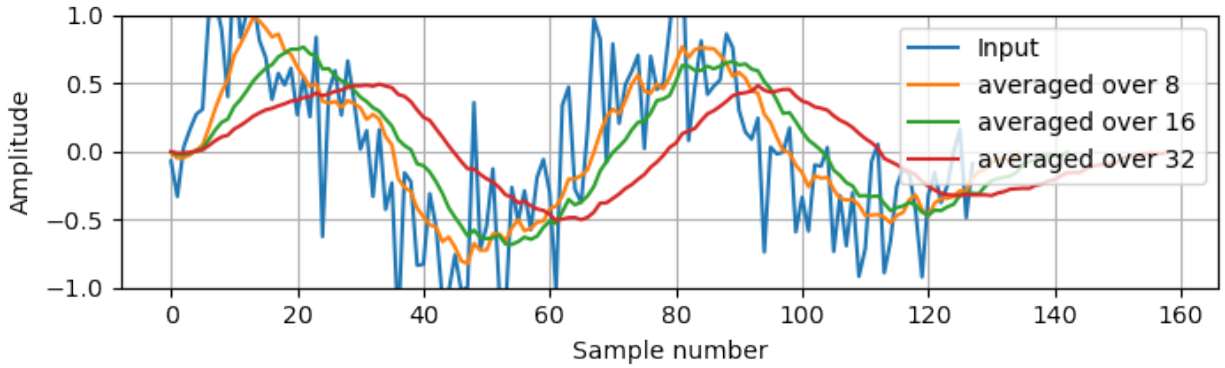

Fig. 2.16: MA algorithm in removing noise

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.

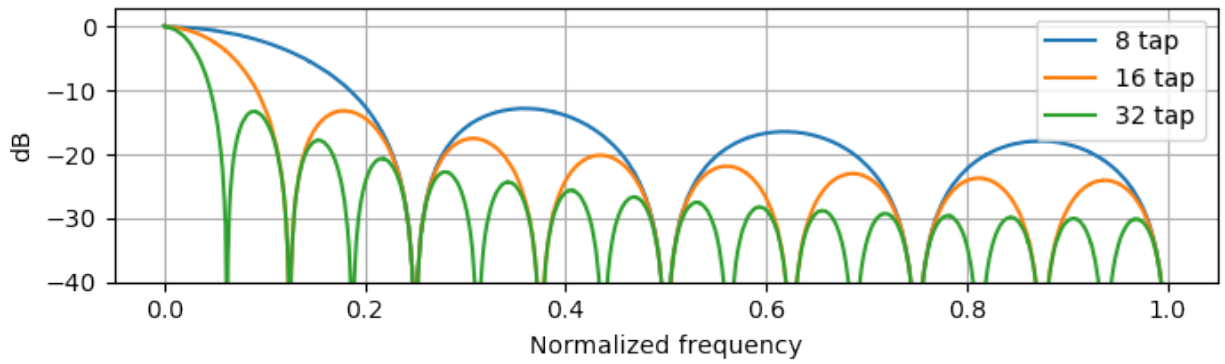

Fig. 2.17: Frequency response of MA filter

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.

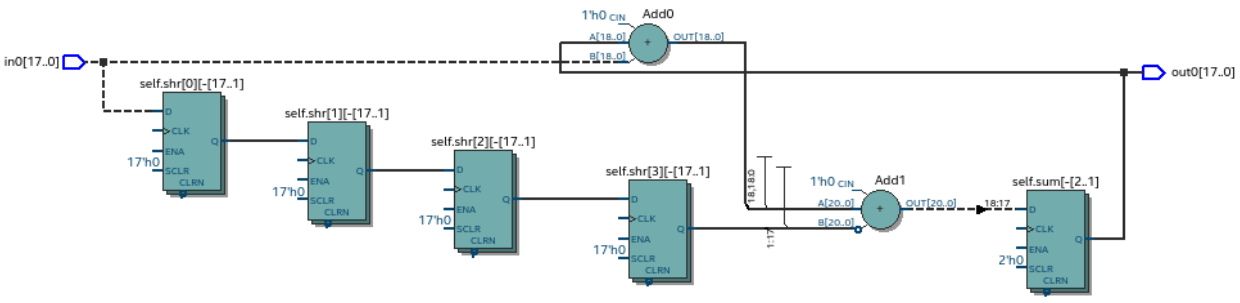

Fig. 2.18: RTL view of moving average (Intel Quartus RTL viewer)

Simulation/Testing

MA is an optimal solution for performing matched filtering of rectangular pulses *[26]*. 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.

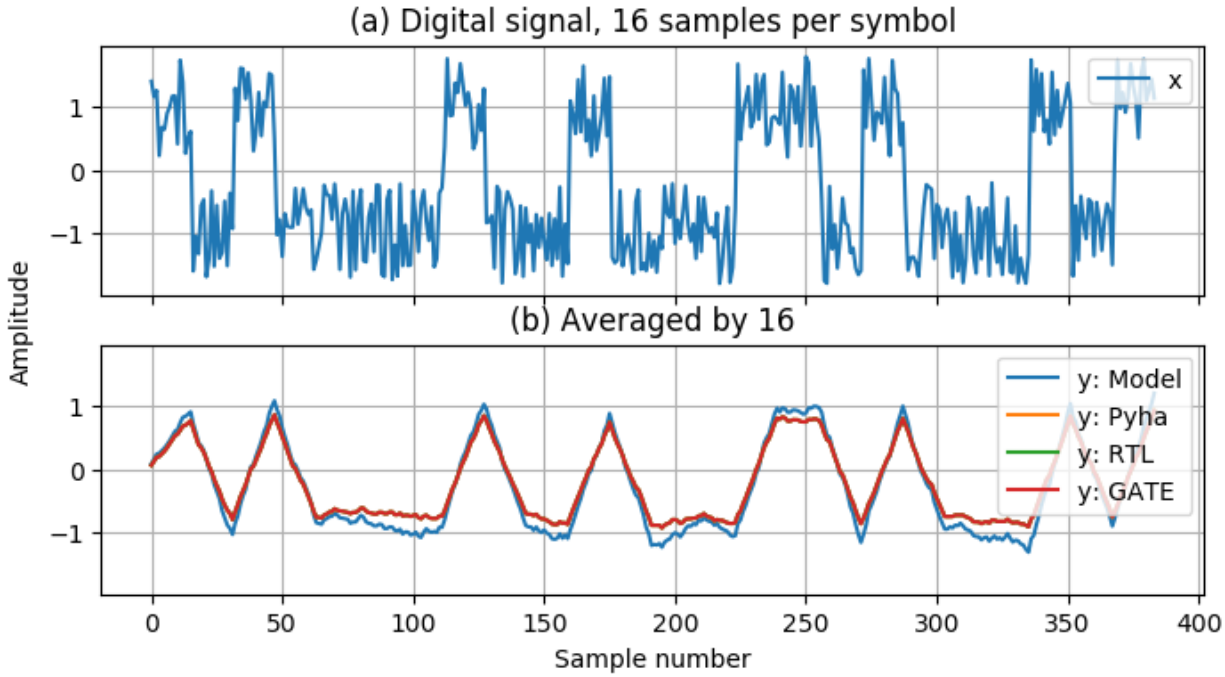

Fig. 2.19: Moving average as matched filter

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 *[27]*.

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 *[28]*, 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) *[28]*.

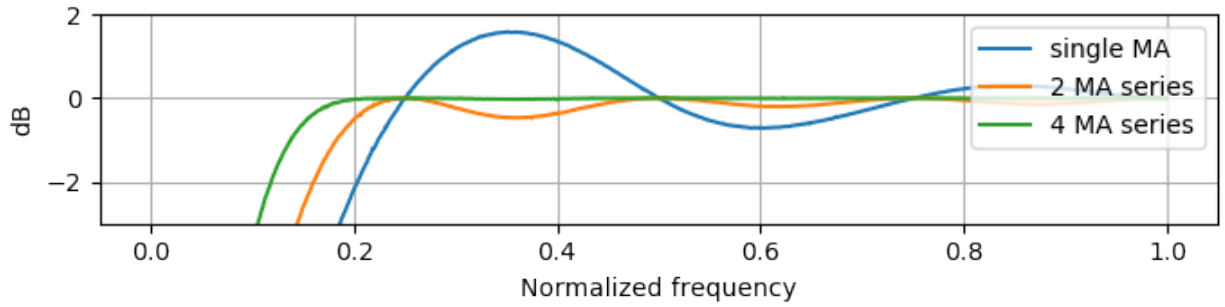

Fig. 2.20: Frequency response of DC removal filter (MA window length is 8)

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 *[28]*. 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.

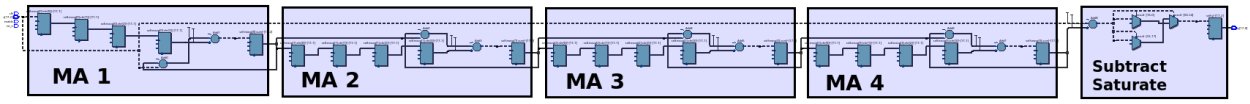

Fig. 2.21: Synthesis result of `DCRemoval(window_len=4)` (Intel Quartus RTL viewer)

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 *[28]*. Another benefit is that this filter delays the signal by only 1 sample.

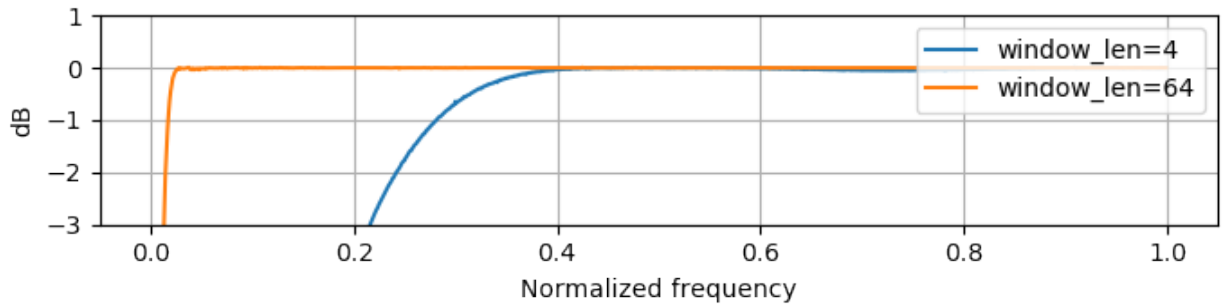

Fig. 2.22: Comparison of frequency response

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.

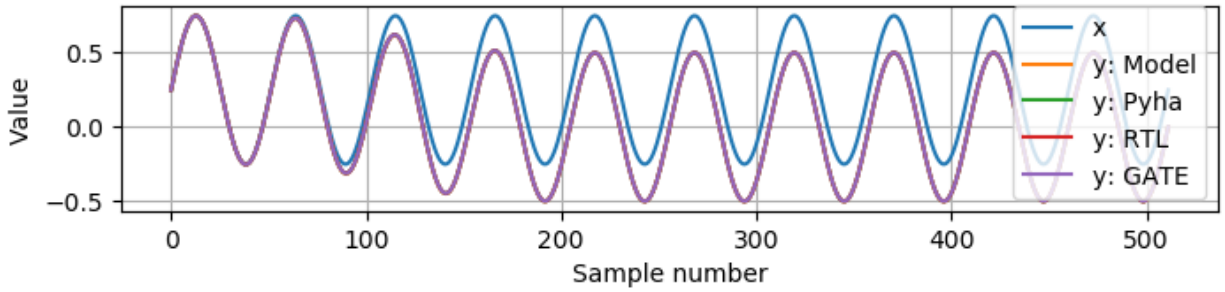

Fig. 2.23: Simulation of DC-removal filter in the time domain

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.

3.1 Introduction

Todo

Pilt converterist

- Simulatsioon tüüpide leidmiseks
- Python AST modifikatsioonid
- Pythont to VHDL
- Sequential OOP VHDL IR

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

3.2 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.

Sequential programming in VHDL has been studied by Jiri Gaislter in *[1]*. 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.

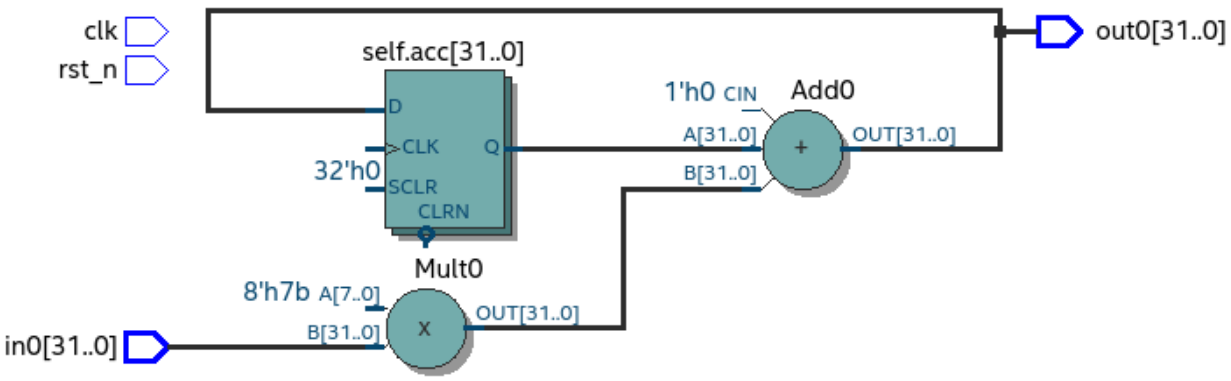

Fig. 3.1: Unexpected synthesis result of Listing 3.1 (Intel Quartus RTL viewer)

3.2.1 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 *[12]*. The same semantics are also used in MyHDL signal objects *[13]*.

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;
```

The last thing that must be handled is loading the **next** to **current**. As stated before, this is done automatically by VHDL for signal assignment; by using variables we have to take care of this ourselves. 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. It is possible to infer multi-clock systems by updating a subset of registers at a different clock edge.

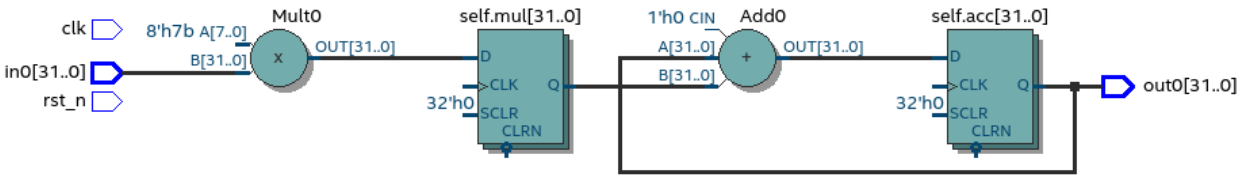

Fig. 3.2: Synthesis result of the revised code (Intel Quartus RTL viewer)

Fig. 3.2 shows the synthesis result of the source code shown in Listing 3.3. It is clear that this is now equal to the system presented at the start of this chapter.

3.2.2 Creating instances

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 *[14]*. 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.

3.2.3 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. A package in VHDL must contain an declaration and body (same concept as header and source files in C).

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

Listing 3.7: Package 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;

    procedure reset(self: inout self_t);
    procedure update_registers(self: inout self_t);
    procedure main(self:inout self_t);
    -- other user defined functions
end package;

package body MAC is
    procedure reset(self: inout self_t) is
    begin
        ...
    end procedure;

    procedure update_registers(self: inout self_t) is
    begin
        ...
    end procedure;

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

3.2.4 Example: Instances in series

Creating a new class that connects two MAC instances in series is simple, first we need to create two MAC instances called MAC_0 and MAC_1 and add them to the data model (Listing 3.8).

Listing 3.8: Datamodel of 'series' class, in OOP-style VHDL

```
type self_t is record
    mac0: MAC_0.self_t;
    mac1: MAC_1.self_t;

    nexts: next_t;
end record;
```

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 (Listing 3.9).

Listing 3.9: Function that connects two MAC's in series, in OOP-style VHDL

```
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_1.main(self.mac1, out_tmp, ret_0=>ret_0);
end procedure;
```

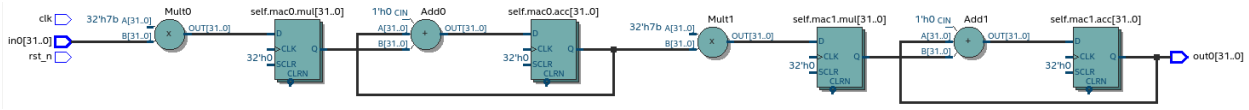

Fig. 3.3: Synthesis result of the new class (Intel Quartus RTL viewer)

Logic is synthesized in series (Fig. 3.3). That is exactly what was specified.

3.2.5 Example: Instances in parallel

Connecting two MAC's in parallel can be done by just returning output of MAC_0 and MAC_1 (Listing 3.10).

Listing 3.10: 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);
    MAC_1.main(self.mac1, a, ret_0=>ret_1);
end procedure;
```

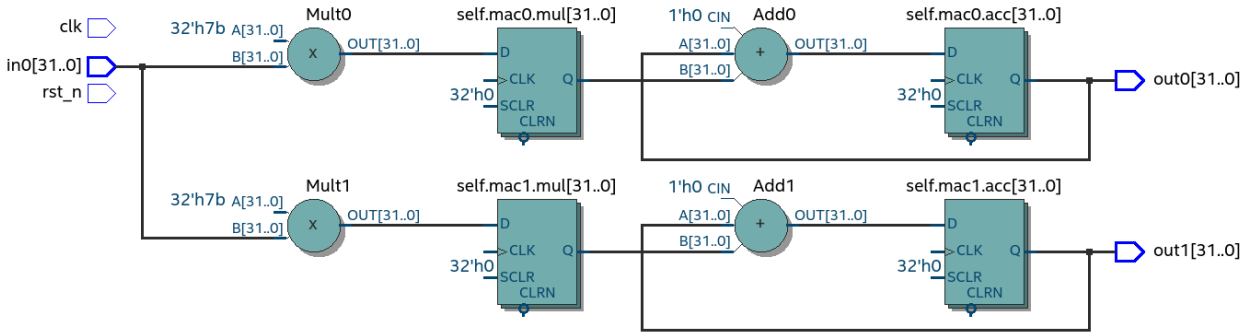

Fig. 3.4: Synthesis result of Listing 3.10 (Intel Quartus RTL viewer)

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

3.2.6 Example: Parallel instances in different clock domains

Multiple clock domains can be easily supported by updating registers at specified clock domains. Listing 3.11 shows the contents of a top-level process, where 'mac0' is updated by 'clk0' and 'mac1' by 'clk1'. Note that nothing has to be changed in the data model or main function.

Listing 3.11: 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);
    end if;

    if rising_edge(clk1) then
        MAC_1.update_registers(self.mac1);
    end if;
end if;
```

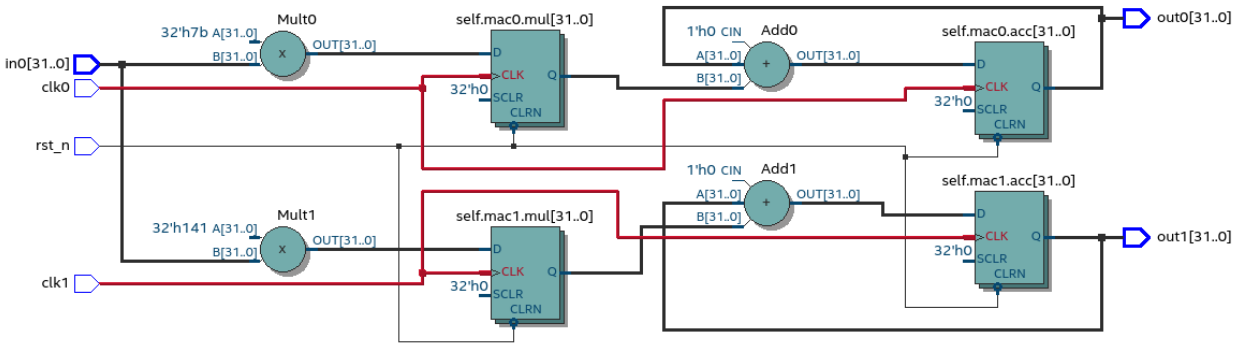

Fig. 3.5: Synthesis result with modified top-level process (Intel Quartus RTL viewer)

Synthesis result (Fig. 3.5) is as expected, MAC's are still in parallel but now the registers are clocked by different clocks. The reset signal is common for the whole design.

3.3 Conversion methodology

Conversion process is based heavily on the results of last chapter, that developed OOP style for VHDL. This simplifies the conversion process in a way, that mostly no complex conversions are not needed. Basically the converter should only care about syntax conversion, that is Python syntax to VHDL.

Thats why this can be called Python bindings.. everything you write in Python has a direct mapping to VHDL, most of the time mapping is just syntax difference.

Still converting Python syntax to VHDL syntax poses some problems. First, there is a need to traverse the Python source code and convert it. Next problem is the types, while VHDL is strongly types language, Python is not, somehow the conversion progress should find out all the types.

This chapter deals with these problems.

This chapter aims to convert the Python based model into VHDL, with the goal of synthesis.

3.3.1 Problem of types

The biggest challenge in conversion from Python to VHDL is types, namely Python does not have them, while VHDL has.

For example in VHDL, when we want to use local variable, it must be defined with type.

Listing 3.12: VHDL variable action

```
-- define variable a as integer
variable a: integer;

-- assign 'b' to 'a', this requires that 'b' is same type as 'a'
a := b;
```

Listing 3.13: Python variable action

```
# assign 'b' to 'a', 'a' will inherit type of 'b'
a = b
```

Listing 3.12 and Listing 3.13 show the variable difference in VHDl and Python. In general this can be interpreted in a way that VHDL icludes all the information required but Python leaves some things open. In Python it is even possible that 'a' is different type for different function callers. Python way is called dynamic-typing while VHDl way is static. Dynamic, meaning that types only come into play when the code is executing.

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 also the code readability suffers.

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 tackle this problem is to request the user to provide top level input types on conversion. As the main types are known, clearly all other types can be derived from them. 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', also the the assign could be part of expression like `a = b < 1`, anyhow this solution gets complex really fast and is not feasible option.

Alternative would be to embrace the dynamic typing of Python and simulate the design before conversion, in that way all the variables resolve some type, thanks to running the code.

Class

Class variables are easy to infer after code has been executed as all of them can be readily accessed.

Listing 3.14: Type problems

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

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

Class variables types can be extracted even without 'simulation'. On class creation '__init__' function runs that also assigns something to all class variables, that is enough to determine type. Still simulation can help Lazy types to converge.

Example:

Listing 3.15: Class variable type

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

Listing 3.15 show example for getting the type of class variable. It initializes the class with argument 5, that is passed to the 'coef' variable. After Python 'type' can be used to determine the variable type. Clearly this variables could be converted to VHDL 'integer' type (not really...Python is infinite).

Locals

Locals mean here the local variables of a function including the function arguments, in VHDL these also require to be typed.

Inferring the type of function local variables is much harder as Python provides no standard way of doing so. This task is hard as locals only exsist in the stack, thus they will be gone once the function exection is done. Luckly this problem has been encountered before in *[15]*, whicp provides an solution.

This approach works by defining a profile tracer function, which has access to the internal frame of a function, and is called at the entry and exit of functions and when an exception is called. *[15]*

Solution is to wrap the function under inspection in other function that sets a traceback function on the return and saves the result of the last locals call.

That way all the locals can be found on each call. Pyha uses this approach to keep track of the local values. Below is an example:

Listing 3.16: Function locals variable type

```
>>> dut.main.locals # before any call, locals are empty
{}
>>> 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'>
```

Advantages

Major advantage of this method is that the type info is extracted easily and complexity is low. Potential perk in the future is that this way could keep track of all values that any variable takes during the simulation, this will be essential if in the future some automatic float to fixed point compiler is to be implementend.

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

Another advantage is that programming in Python can be even more lazy..

Disadvantages

Downside of this solution is obviously that the desing must be simulated in Python domain before it can be converted to VHDL. First clear is that the design must be simulated in Python domain before conversion is possible, this may be inconvenient.

Also the simulation data must cover all the cases, for example consider the function with conditional local variable, as shown on `cond-main`. If the simulaton passes only True values to the function, value of variable 'b' will be unknown ad vice-versa. Of course such kinf of problem is detected in the conversion process. Also in hardware we generally have much less branches than in software also all of thes branches are likely to be important as each of them will **always** take up resources.

Listing 3.17: Type problems

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

3.3.2 Conversion methodology

After the type problem has been solved, next step is to convert the Python code into VHDL.

Chapter *Conversion to VHDL* developed a way to write OOP VHDL, thanks to this, the conversion from Python to VHDL is much simplified. Mostly the converter needs to convert

the syntax parts. Conversion progress requires no understanding of the source code nor big modifications.

This task requires a way of parsing the input Python code, making modifications and then outputting VHDL compilant syntax.

In general this step involves using an abstract syntax tree (AST). This reads in the source file and turns it into traversable tree stucture of all the operations done in the program.

There are many tools in the Python ecosystem that allow this task, for example lib2to3 etc.

Converter of this project uses the RedBaron *[16]*. RedBaron is an Python library with an aim to significantly simply operations with source code parsing.

RedBaron is a python library with intent of making the process of writing code that modify source code as easy and as simple as possible. That include writing custom refactoring, generic refactoring, tools, IDE or directly modifying you source code into IPython with a higher and more powerful abstraction than the advanced texts modification tools that you find in advanced text editors and IDE. *[16]*

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

Simple example of RedBaron operation is shown on Listing 3.18. It uses a simple `a = 5` assigment as the input and shows how RedBaron turns the code into special 'nodes'.

Listing 3.18: Radbaron output for `a = 5`

```
>>> red = RedBaron('a = 5')
>>> red.help()
0 -----------------------------------------------------
AssignmentNode()
  # identifiers: assign, assignment, assignment_, assignmentnode
  operator=''
  target ->
    NameNode()
      # identifiers: name, name_, namenode
      value='a'
  value ->
    IntNode()
      # identifiers: int, int_, intnode
      value='5'
```

It shows that the input code is turned into 'AssigmentNode' object, that has 3 parameters:

- Operator -
- Target - assignment target
- Value - value assigned to target

The power of RedBaron is that, these objects can be very easly modified. For example, one could set `red[0].value = '5 + 1'` and this would turn the overall code to `a = 5 + 1`. RedBaron also provides methods to, for example 'find' can be used to find all the 'assignment' nodes in the code.

Pyha handles the conversion to VHDL by overwriting the RedBaron nodes. For example for the 'AssignmentNode' Pyha inherits from the base node but changes the string output so that assignment operator '=' is changed to ':=' and at the end of the expression ';' is added. So the output would be `a := 5;`, that is VHDL compatible statement.

For example in the above example main node is AssignmentNode, this could be modified to change the '=' into ':=' and add ';' to the end of line. Resulting in a VHDL compatible statement `a := 5;`.

3.3.3 Basic conversions

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

3.3.4 Converting functions

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

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

VHDL supports two style of functions:

- Functions - classical functions, that have input values and can return one value
- Procedures - these cannot return a value, but can have agument that is of type 'out', thus returing trough an output argument. Also it allows argument to be of type 'inout' that is perfect for class object.

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

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

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

- First argument self
- Input argument

- Local variables
- Multiple return values

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

Listing 3.19: VHDL example procedure

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

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

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

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

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

3.4 Summary

This chapter presented the proposed, fully synthesizable, object-oriented model for VHDL.

Its major advantage is that none of the VHDL data-flow semantics are used (except for top level entity). This makes development similar to regular software. Programmers new to the VHDL language can learn this way much faster as their previous knowledge of other languages transfers.

Moreover, this model is not restricted to one clock domain and allows simple way of describing registers.

The major motivation for this model was to ease converting higher level languages into VHDL. This goal has been definitely reached, next section of this thesis develops Python bindings with relative ease. Conversion is drastically simplified as Python class maps to VHDL class, Python function maps to VHDL function and so on.

Todo

Careful. You have only used relatively simple examples. To say 'definitely reached' you should have substantial evidence based on a large number of cases and/or some sort of formal proof.

Synthesizability has been demonstrated using Intel Quartus toolset. Bigger designs, like frequency-shift-keying receiver, have been implemented on Intel Cyclone IV device. There has been no problems with hierarchy depth, objects may contain objects which themselves may contain arrays of objects.

Chapter 4

Conclusion

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

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

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

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

Todo

Moving VHDL programmers to this tool? problems?

4.1 Summary

4.2 Limitations/future work

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

Currently designs are limited to one clock signal, decimators are possible by using Streaming interface. Future plans is to add support for multirate signal processing, this would involve

automatic PLL configuration. I am thinking about integration with Qsys to handle all the nasty clocking stuff.

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

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

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

The initial goal of Pyha was to test ou how well could the software approach apply to the hardware world. As this thesis shows that it is working well, the generated hardware output is unexpected to software people but resulting output is the same. Pyha is an exploratory project, many things work and ca be done but still much improvements are needed for example, inclusion of bus models like Wishbone, Avalon, AXI etc. Also currently Pyha works on single clock designs, while its ok because mostly today desings are just many single clock designs connected with buses.

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[15] 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-/.

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

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

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

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

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

[21] 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/.

[22] Clifford Wolf. Yosys open synthesis suite. URL: http://www.clifford.at/yosys/.

[23] Florian Mayer. A vhdl frontend for the open-synthesis toolchain yosys. Master's thesis, Hochschule Rosenheim, 2016.

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

[25] 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.

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

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

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