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

Pyha

Pyha is an tool that allows writing of digital hardware in Python language. Currently it focuses mostly on the DSP applications.

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'_**)
- Decent quality VHDL conversion output (get what you write, keeps hierarchy)
- Integration to Intel Quartus (run GATE level simulations)
- Tools to simplify verification

Pyha specifically focuses on making testing of the DSP algorithms simpler. While many alternatives are based on C language, but most of the hardware design time is used up in the process of testing and verification, who would like to do this in C, Python is much better language for this!

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.

Migen cannot be debugged, this may not seem like a big upside for Pyha. But it is, steping trough the code can greatly simplify finding bugs, after all this is the main way of debugging in conventional programming. Also debugger is useful tool for understanding the codebase.

1.1 Introduction

This chapter focuses on the Python side of Pyha, while the next chapter gives details on how Pyha details are converted to VHDL and how they can be synthesised.

A multiply-accumulate(MAC) circuit is used as a demonstration circuit throughout the rest of this chapter. It is a good choice as it is powerful element yet not very complex. Last chapter of this thesis peresents more serious use cases.

Note: The first half of this chapter uses ‘integers’ as base type in order to keep the examples simple. Second half starts using fixed-point numbers, that ade default for Pyha.

1.2 Model based design

Generally before the hardware system is implemented, it is useful to first experiment with the idea and maybe even do some performance figures like SNR. For this, model is constructed. In general the model is the simplest way to archive the task, it is not optimized.

Model allows to focus on the algorithmical side of things. Also model comes in handy when verifying the operation of the hardware model. Output of the model and hardware can be compared to verify that the hardware is working as expected.

In *[2]*, 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) *[3]*. Using this design flow, users describe their designs in Simulink using blocks provided by Xilinx System Generator *[3]*.

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.

While this workflow is powerful indeed.

Model based design, this is also called behavioral model (.. https://books.google.ee/books?hl=en&lr=&id=XbZr8DurZYEC&oi=fnd&pg=PP1&dq=vhdl&ots=PberwiAymP&sig=zqc4BUSmFZaL3hxRilU-J9Pa_5I&redir_esc=y#v=onepage&q=vhdl&f=false)

1.2.1 Pyha flow

Pyha is fully open-source software, meaning it is a free tool to use by anyone. Since Pyha is based on the Python programming language, it gets all the goodness of this environment.

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

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.

VHDL uuendused? VUNIT VUEM?

Test-driven development / unit-tests

Model based development How MyHDl and other stuffs contribute here?

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

Listing 1.1: Multiply-accumulate written in Python

```python
class MAC:
    def __init__(self, coef):
        self.coef = coef

    def model_main(self, sample_in, sum_in):
        import numpy as np

        muls = np.array(sample_in) * self.coef
        sums = muls + sum_in
        return sums
```

Listing 2.1 shows the MAC model written in Python. It uses the Numpy package for numeric calculations.

1.3 Testing/debugging and verification

1.3.1 Simplifying testing

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

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

- Siin all ka unit testid?

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

Siin peaks olema test funksioonid?

Ipython testing...show example with two unit tests and plots.

1.4 Describing hardware

Assuming we have now enough knowledge and unit-tests we can start implementing the Hardware model.

Main idea of Pyha is to enable hardware design in Python ecosystem.

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

Listing 1.2: Basic Pyha unit

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

    def main(self, input):
        pass

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

Listing 1.2 shows the besic design unit of the developend tool, it is a standard Python class, that is derived from a baseclass *HW, purpos of this baseclass is to do some metaclass stuff and register this class as Pyha module.

Metaclass actions:

1.4.1 Stateless logic

Clock abstracted as forever running loop. In hardware determines how long time we need to wait before next call to function so that all signals can propagate.

Stateless is also called combinatory logic. In the sense of software we could think that a function is stateless if it only uses local variables, has no side effects, returns are based on inputs only. That is, it may use local variables of function but cannot use the class variables, as these are stateful.

Listing 1.3: Stateless MAC implemented in Pyha

```
class MAC(HW):
    def main(self, x, sum_in):
        mul = 123 * x
        y = sum_in + mul
        return y

    def model_main ...
```

Listing 1.3 shows the design of a combinatory logic. In this case it is a simple xor operation between two input operands. It is a standard Python class, that is derived from a baseclass *HW, purpose of the baseclass is to do some metaclass stuff and register this class as Pyha module.

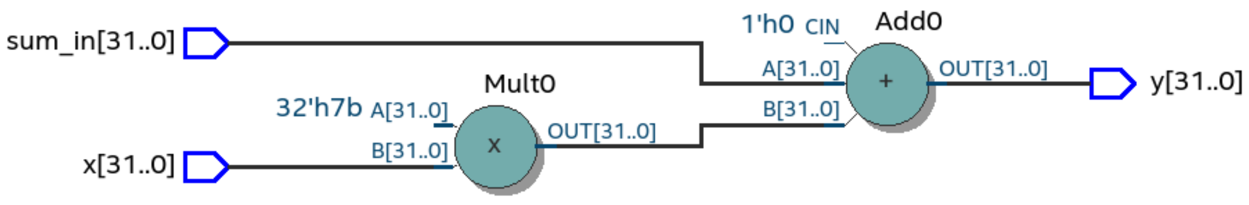

Fig. 1.1: Synthesis result of the revised code (Intel Quartus RTL viewer)

Fig. 2.5 shows the synthesis result of the source code shown in Listing 2.8. It is clear that this is now equal to the system presented at the start of this chapter.

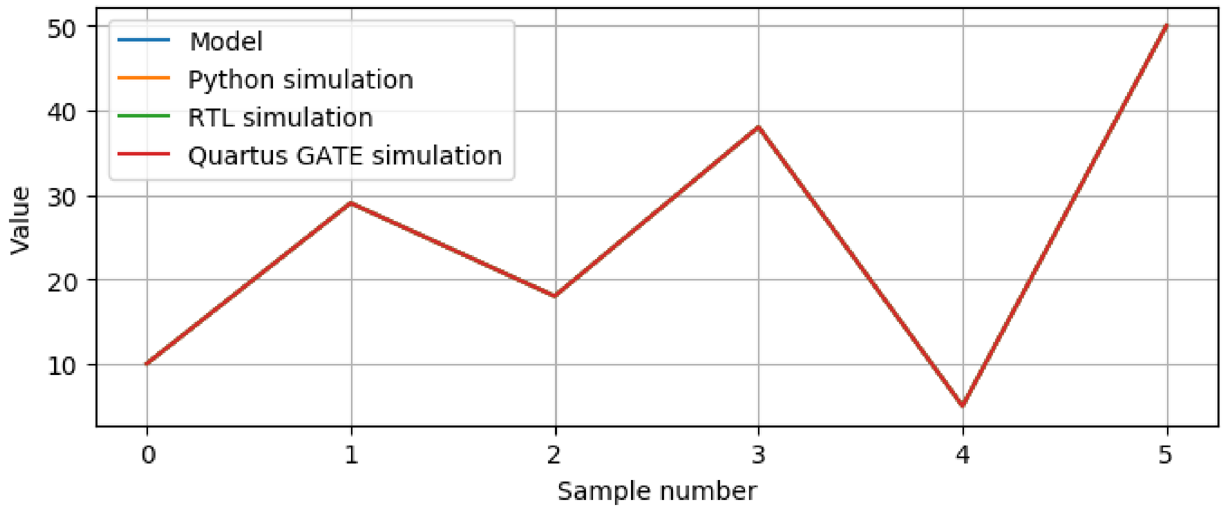

Fig. 1.2: Synthesis result of the revised code (Intel Quartus RTL viewer)

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

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

Todo

how this turns to VHDL and RTL picture?

In software operations consume time, but in hardware they consume resources, general rule.

Not clocked...basically useless analog stuff.

1.4.2 Sequential logic

1.4.3 Understanding registers

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.

In traditional programming, class variables are very similar to local variables. The difference is that class variables can 'remember' the value, while local variables exist only during the function execution.

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`.

Jan Decaluwe, the author of MyHDL package, has written a relevant article about the necessity of signal assignment semantics *[4]*.

Using an signal assignment inside a clocked process always infers a register, because it exactly represents the register model.

Registers in hardware have more purposes:

- delay
- max clock speed - how this corresponds to sample rate?

Explain somwhere that each call to function is a clock tick.

Listing 1.4: Basic sequential circuit in Pyha

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

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

Listing 1.4 shows the design of a registered adder.

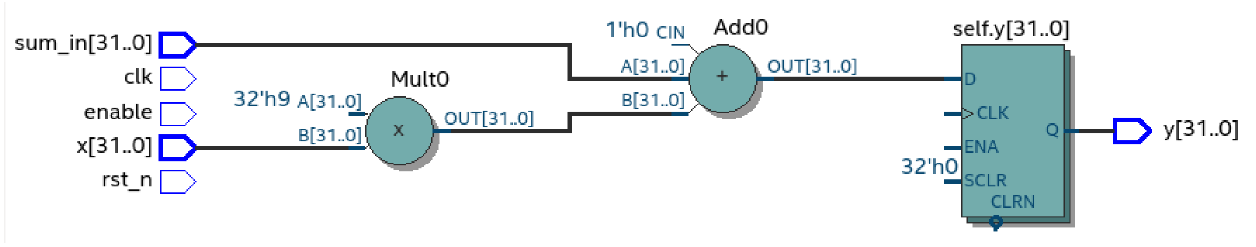

Fig. 1.3: Synthesis result of the revised code (Intel Quartus RTL viewer)

Fig. 1.3 shows the synthesis result of the source code shown in Listing 2.8. It is clear that this is now equal to the system presented at the start of this chapter.

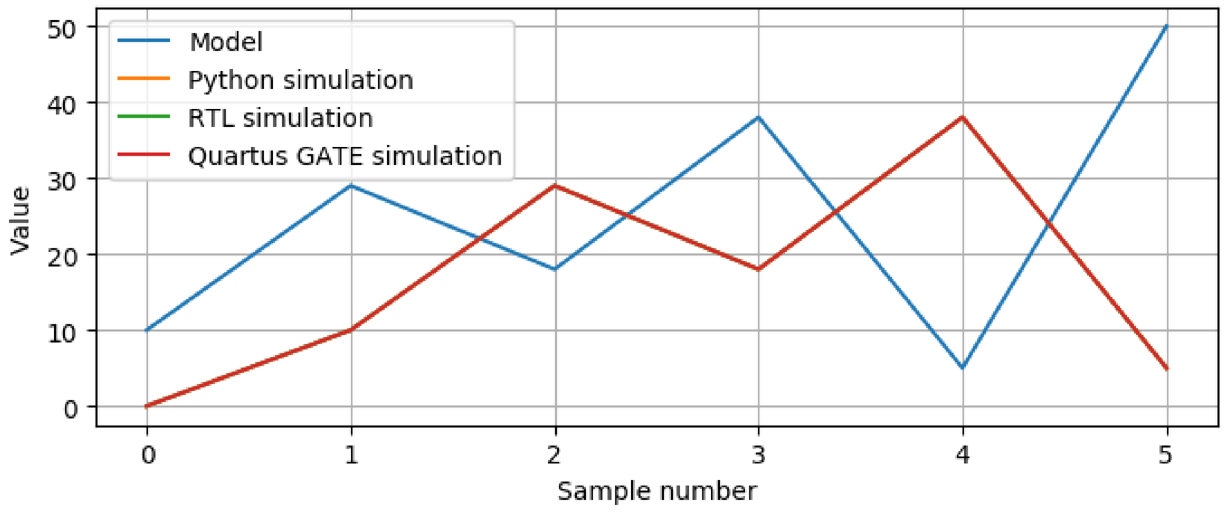

Fig. 1.4: Synthesis result of the revised code (Intel Quartus RTL viewer)

Running the same testing code results in a Fig. 1.4. It shows that while the Python, RTL and GATE simulations are equal, model simulation differs. This is the effect of added register, it adds one delay to the harwdware simulations.

This is an standard hardware behaviour. Pyha provides special variable `self._delay` that specifies the delay of the model, it is useful:

- Document the delay of your blocks
- Upper level blocks can use it to define their own delay
- Pyha simulations will adjust for the delay, so you can easily compare to your model.

Note: Use `self._delay` to match hardware delay against models

After setting the `self._delay = 1` in the __init__, we get:

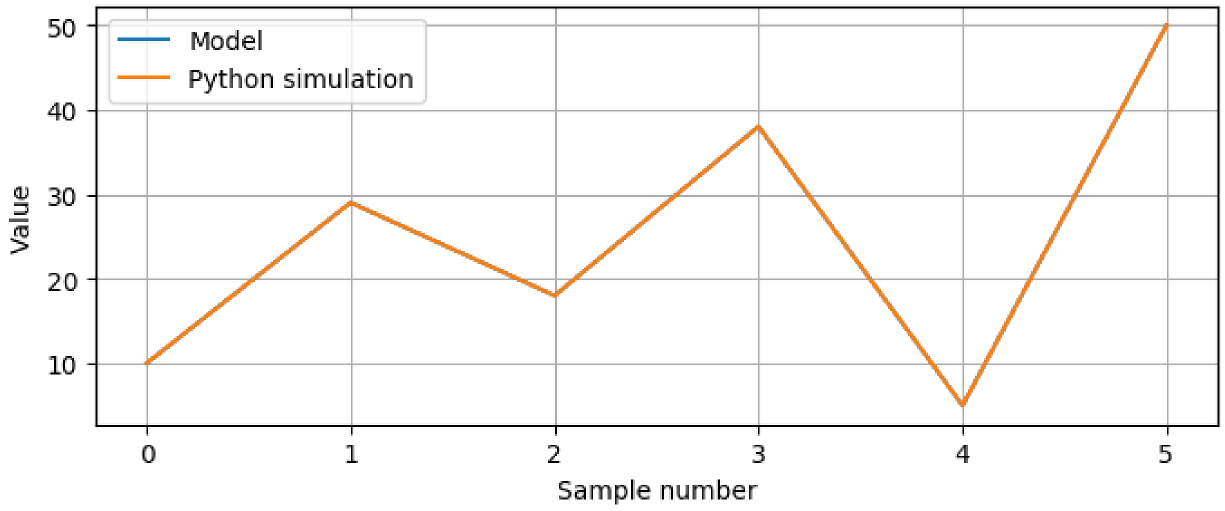

Fig. 1.5: Synthesis result of the revised code (Intel Quartus RTL viewer)

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

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

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

In general this system is similiar to VHDL signals:

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

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

Todo

how this turns to VHDL and RTL picture?

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

Simulation a

1.5 Fixed-point designs

1.6 Extended example

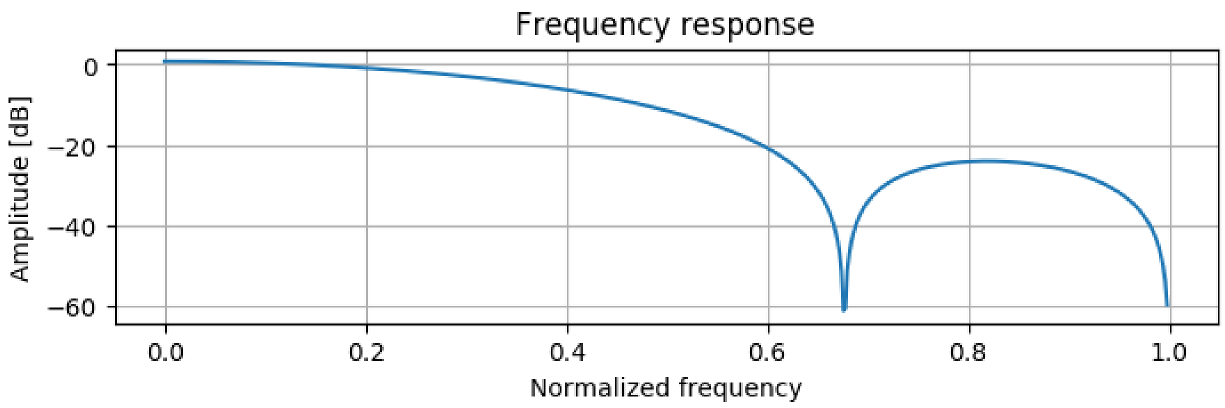

Fig. 1.6: Synthesis result of the revised code (Intel Quartus RTL viewer)

Note that design uses only 2 18 bit multipliers.

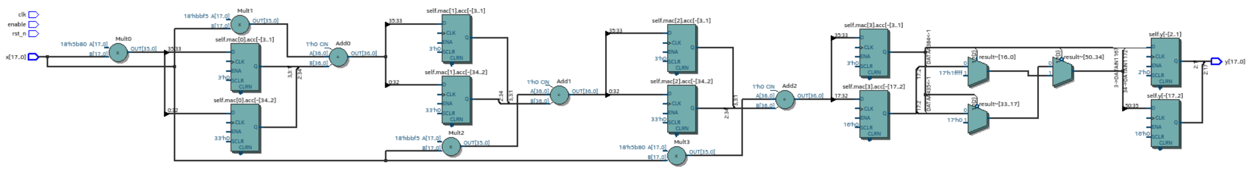

Fig. 1.7: Synthesis result of the revised code (Intel Quartus RTL viewer)

1.7 Conclusions

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

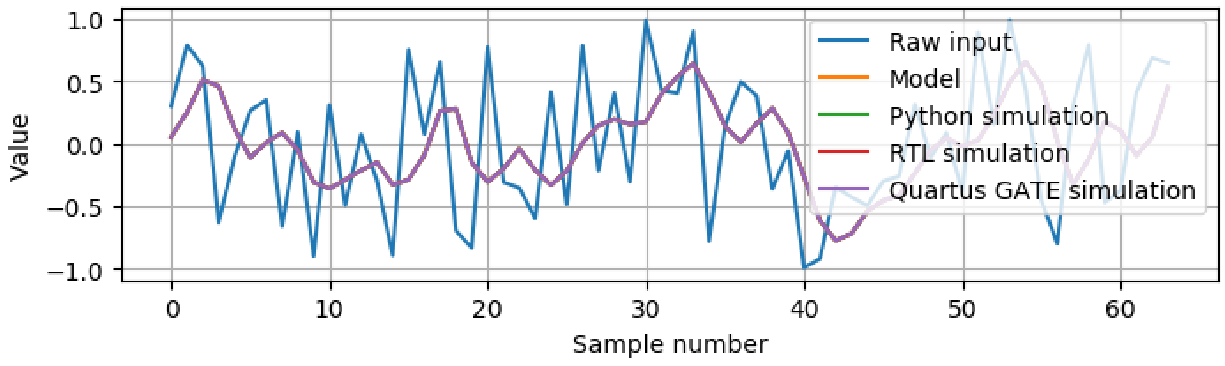

Fig. 1.8: Synthesis result of the revised code (Intel Quartus RTL viewer)

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

Future stuff: Make it easier to use, windows build?

Chapter 2

VHDL as intermediate language

This chapter is a fair bit more techincal and requires some knowledge of VHDL and hardware synthesis.

This chapter develops 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.

2.1 Introduction

2.1.1 Background

The most commonly used design 'style' for synthesizable VHDL models is what can be called the 'dataflow' style. A larger number of concurrent VHDL statements and small processes connected through signals are used to implement the desired functionality. Reading and understanding dataflow VHDL code is often deemed difficult since the concurrent statements and processes do not execute in the order they are written, but when any of their input signals change value *[8]*.

The biggest difference between a program in VHDL and standard programming language such as C, is that VHDL allows concurrent statements and processes that are scheduled for execution by events rather than in then order they are written. This reflects indeed the dataflow behaviour of real hardware, but is difficult to understand and analyse. On the other hand, analysing the behaviour of programs written in sequential programming languages does not become a problem even if the program tends to grow, since execution is done sequentially from top to bottom *[8]*.

Jiri Gaisler has proposed a 'Structured VHDL design method *[8]*' in the ˜2000 .He suggests to raise the hardware design abstraction level by using a two-process method.

The two-process method only uses two processes per entity: one process that contains all combinatory (asynchronous) logic, and one process that contains all sequential logic (registers). Using this structure, the complete algorithm can be coded in sequential (non-concurrent) statements in the combinational process while the sequential process only contains registers, i.e. the state *[8]*.

Todo

Are there publications or other sources that comment/evaluate his approach (i.e. what are the pros and cons?)

Object-oriented style in VHDL has been studied before. In *[9]* a proposal was made to extend the VHDL language with OOP semantics (dataflow based), this effort ended with the development of OO-VHDL *[10]*, a VHDL preprocessor, turning proposed extensions to standard VHDL. This work did not make it the VHDL standard, the status of compiler is unknown, latest publicly available document dates to year 1999.

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.

Todo

Description of the tools?

The author of MyHDL package has written good blog posts about signal assignments *[4]* and software side of hardware design *[11]*. These ideas are relevant for this chapter.

2.1.2 Objective

The main motivation of this work is to use VHDL as an intermediate language for High-Level synthesis.

While the work of Jiri Gaisler greatly simplifies the programming experience of VHDL, it still has some major drawbacks:

- It is applicable only to single-clock designs:cite:*structvhdl_gaisler*;
- The ‘structured’ part can be only used to define combinatory logic, registers must be still inferred by signals assignments;
- It still relies on many of the VHDL dataflow features, for example, design reuse is achieved trough the use of entities and port maps;

This work aims at improving the ‘two process’ model by proposing an Object-oriented approach for VHDL, lifting all the previously listed drawbacks.

This section uses examples in Python language in order to demonstrate the Python to VHDL converter (developed in the next chapter) and set some targets for the intermediate language.

A multiply-accumulate(MAC) circuit is used as a demonstration circuit throughout the rest of this chapter.

Todo

Need to introduce Pyha before.

Listing 2.1: Pipelined multiply-accumulate(MAC) specified in Pyha

```python
class MAC:
    def __init__(self, coef):
        self.coef = coef
        self.mul = 0
        self.acc = 0

    def main(self, a):
        self.next.mul = a * self.coef
        self.next.acc = self.acc + self.mul
        return self.acc
```

Note: In order to keep examples simple, only `integer` types are used in this chapter.

Listing 2.1 shows a MAC component implemented in Pyha (Python to VHDL compiler implemented in the next chapter of this thesis). The purpose of this circuit is to multiply the input with the coefficient and accumulate the result. It synthesizes to logic as shown in Fig. 2.1.

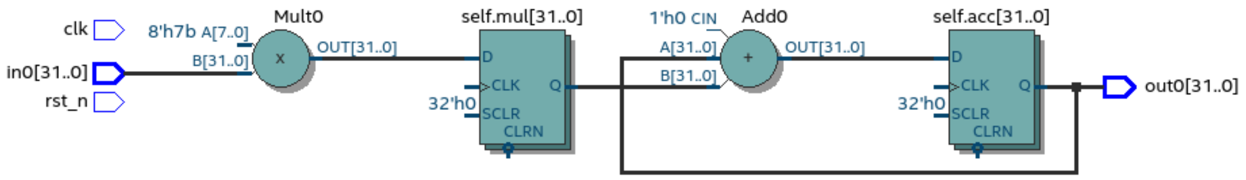

Fig. 2.1: Synthesis result of Listing 2.1 (Intel Quartus RTL viewer)

The main reason to pursue the OOP approach is the modularity and the ease of reuse. Listing 2.2 defines a new class, containing two MACs that are to be connected in series. As expected it synthesizes to a series structure (Fig. 2.2).

Listing 2.2: Two MAC's connected in series, specified in Pyha

```
class SeriesMAC:
    def __init__(self, coef):
        self.mac0 = MAC(123)
        self.mac1 = MAC(321)

    def main(self, a):
        out0 = self.mac0.main(a)
        out1 = self.mac1.main(out0)
        return out1
```

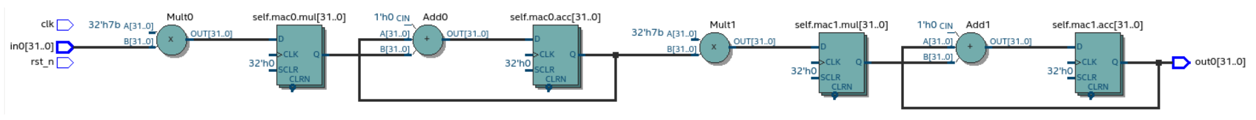

Fig. 2.2: Synthesis result of Listing 2.2 (Intel Quartus RTL viewer)

Todo

Names on the figure should match the names on the code! Explain that 'a' is the input on the left-hand side (fed into B of the 1st MAC), out0 is output of the 1st MAC (fed into B of the 2nd MAC) and 'out1' in the source code is actually out0 in the RTL view (or am I mistaken?)

With slight modification to the 'main' function (Listing 2.3), two MAC's can be connected in a way that synthesizes to a parallel structure (Fig. 2.3).

Listing 2.3: Two MAC's in parallel, specified in Pyha

```
def main(self, a):
    out0 = self.mac0.main(a)
    out1 = self.mac1.main(a)
    return out0, out1
```

It is clear that the OOP style could significantly simplify hardware design. The objective of this work is to develop a synthesizable VHDL model that could easily map to these MAC examples.

Todo

Elaborate on what you mean with 'clear' and 'simplify'.

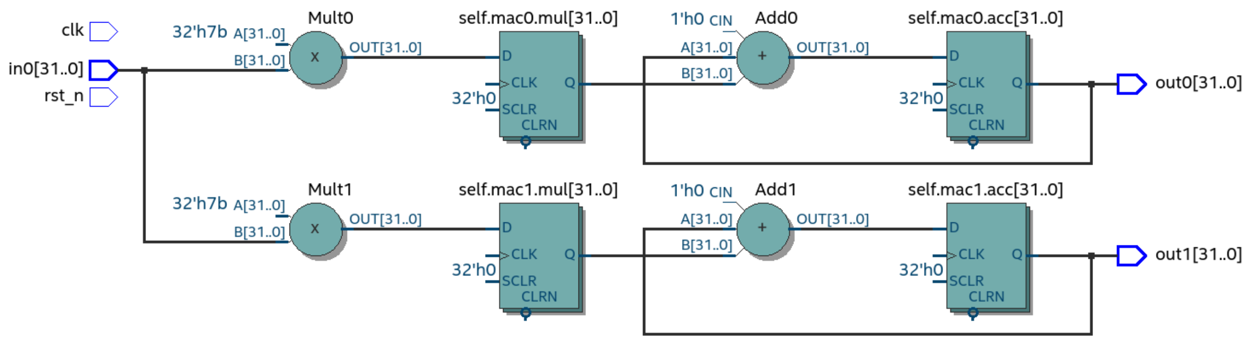

Fig. 2.3: Synthesis result of Listing 2.3 (Intel Quartus RTL viewer)

2.1.3 Using SystemVerilog instead of VHDL

SystemVerilog (SV) is the new standard for Verilog language, it adds significant amount of new features to the language [12]. 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 [13]. 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 [14], 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 [15].

Todo

What is the VHDL frontend status?

2.2 Object-oriented style in VHDL

Todo

Remind the reader that what follows is your proposal (one of the thesis contributions). Also briefly explain what is done differently as compared to previous approaches (especially those that you cited earlier).

While VHDL is mostly known as a dataflow language, it inherits strong support for structured programming from ADA.

Todo

Need to reference that statement.

The basic idea of OOP is to bundle up some common data and define functions that can perform actions on it. Then one could define multiple sets of the 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 includes a 'class' like structure called 'protected types' *[16]*, unfortunately these are not meant for synthesis. Even so, OOP style can be imitated, by combining data in records and passing them as a parameters to 'class functions'. This is essentially the same way how C programmers do it.

Listing 2.4: MAC data model in VHDL

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

Constructing the data model for the MAC example can be done by using VHDL 'records' (Listing 2.4). In the sense of hardware, we expect that the contents of this record will be synthesised as registers.

Note: We label the data model as 'self', to be equivalent with the Python world.

Listing 2.5: OOP style function in VHDL (implementing MAC)

```vhdl
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;
```

An OOP style function can be constructed by adding a first argument that points to the data model object (Listing 2.5). In VHDL, procedure arguments must have a direction, for example the first argument 'self' is of direction 'inout', this means it can be read and also written to.

One drawback of VHDL procedures is that they cannot return a value, instead 'out' direction arguments must be used. The advantage of this is that the procedure may 'output/return' multiple values, as can Python functions.

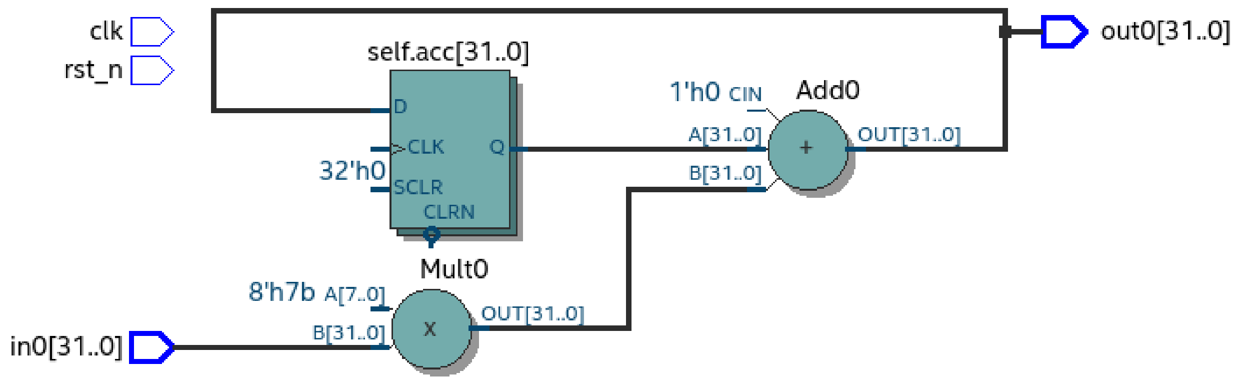

Fig. 2.4: Synthesis result of Listing 2.5 (Intel Quartus RTL viewer)

The synthesis results (Fig. 2.4) 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.

In fact, the signal path from **in0** to **out0** contains no registers at all, making this design hard to use in real designs.

2.2.1 Understanding registers

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.

In traditional programming, class variables are very similar to local variables. The difference is that class variables can 'remember' the value, while local variables exist only during the function execution.

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`.

Jan Decaluwe, the author of MyHDL package, has written a relevant article about the necessity of signal assignment semantics *[4]*.

Using an signal assignment inside a clocked process always infers a register, because it exactly represents the register model.

2.2.2 Inferring registers with variables

While 'signals' and 'signal assignment' are the VHDL way of defining registers, they pose a major problem because they are hard to map to any other language than VHDL. This work aims to use variables instead, because they are the same in every other programming language.

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 *[17]*. The same semantics are also used in MyHDL *[4]*.

Adapting this style for the OOP data model is shown on Listing 2.6.

Listing 2.6: Data model with **next**, in OOP-style VHDL

```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;
end record;
```

The new data model allows reading the register value as before and extends the structure to include the 'nexts' object, so that it can used to assign new value for registers, for example `self.nexts.acc := 0`.

Integration of the new data model to the 'main' function is shown on Listing 2.7. The only changes are that all the 'register writes' go to the 'nexts' object.

Listing 2.7: Main function using 'nexts', in OOP-style VHDL

```
procedure main(self: inout self_t; a: integer; ret_0: out integer) is
begin
    self.nexts.mul := a * self.coef;
    self.nexts.acc := self.acc + self.mul;
    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 2.8 defines new function 'update_registers', taking care of this task.

Listing 2.8: 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.

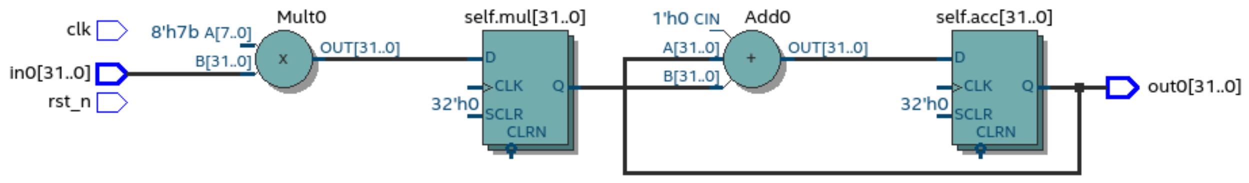

Fig. 2.5: Synthesis result of the revised code (Intel Quartus RTL viewer)

Fig. 2.5 shows the synthesis result of the source code shown in Listing 2.8. It is clear that this is now equal to the system presented at the start of this chapter.

2.2.3 Creating instances

The general approach of creating instances is to define new variables of the 'self_t' type, Listing 2.9 gives an example of this.

Listing 2.9: 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 2.10 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 2.10: 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 *[16]*. 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 2.11).

Listing 2.11: 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.

2.2.4 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 2.12 shows the template package for VHDL 'class'. All the class functionality is now in one common namespace.

Listing 2.12: 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;
```

2.3 Examples

This section provides some simple examples based on the MAC component and OOP model, that were developed in previous chapter.

2.3.1 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 2.13).

Listing 2.13: 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 2.14).

Listing 2.14: 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;
```

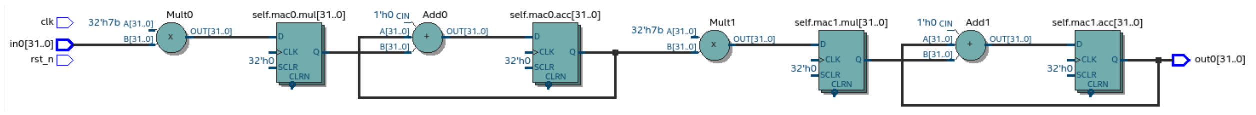

Fig. 2.6: Synthesis result of the new class (Intel Quartus RTL viewer)

Logic is synthesized in series (Fig. 2.6). That is exactly what was specified.

2.3.2 Instances in parallel

Connecting two MAC's in parallel can be done by just returning output of MAC_0 and MAC_1 (Listing 2.15).

Listing 2.15: 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;
```

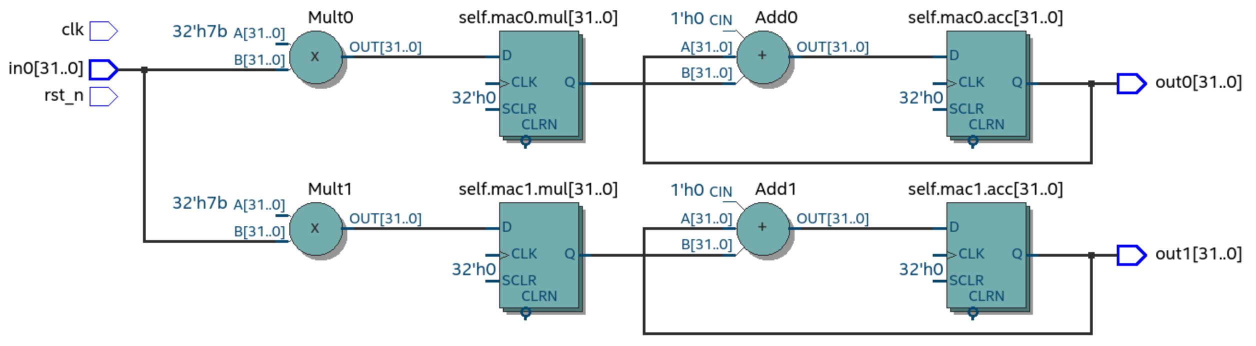

Fig. 2.7: Synthesis result of Listing 2.15 (Intel Quartus RTL viewer)

Two MAC's are synthesized in parallel, as shown in Fig. 2.7.

2.3.3 Parallel instances in different clock domains

Multiple clock domains can be easily supported by updating registers at specified clock domains. Listing 2.16 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 2.16: Top-level for multiple clocks, in OOP-style VHDL

```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;
```

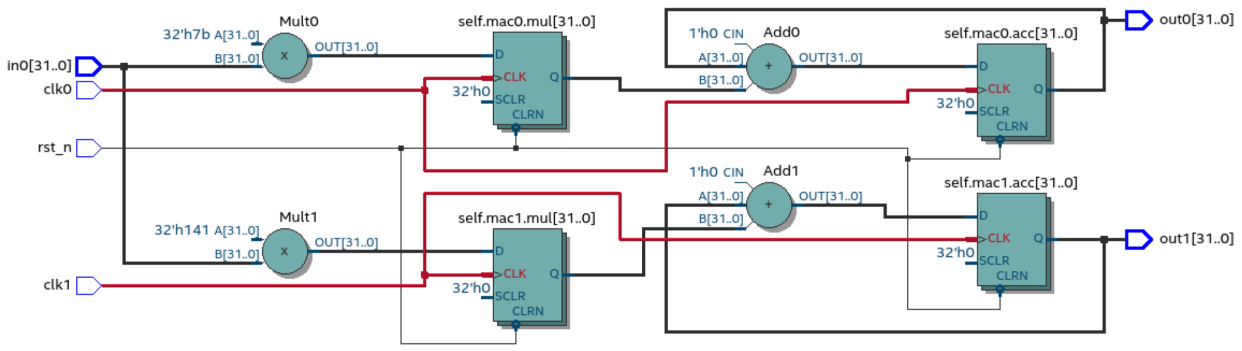

Fig. 2.8: Synthesis result with modified top-level process (Intel Quartus RTL viewer)

Synthesis result (Fig. 2.8) 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.

Todo

Add TDA example here? Would demonstrate statemechines and control structures...

2.4 Conclusion

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.

Bibliography

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

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

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

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

[5] 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-/.

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

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

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

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

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

[11] Jan Decaluwe. Thinking software at the rtl level. URL: http://www.jandecaluwe.com/hdldesign/thinking-software-rtl.html.

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

[13] 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/.

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

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

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

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