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

Gaspar Karm

VHDL as Object-oriented Intermediate Language and Python bindings with simulator

Master's Thesis

Supervisors:
Muhammad Mahtab Alam
PhD
COEL ERA-Chair,
Associate Professor

Yannick Le Moullec
PhD
Professor

Contents:

Chapter 1

VHDL as intermediate language

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

1.1 Introduction

1.1.1 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 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. *[4]*

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 *[4]*.

Jiri Gaisler has proposed an 'Structured VHDL design method *[4]*' in the ˜2000 .He suggest to raise the hardware design abstraction level by using 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 *[4]*.

Object-oriented style in VHDL has been studied before. In *[5]* proposal was made to extend VHDL language with OOP semantics (dataflow based), this effort ended with development of OO-VHDL *[6]*, an VHDL preprocessor, turning proposed extensions to standard VHDL. This work did not make it do VHDL standard, the status of compiler is unknown.

Many tools on the market are capable of convert 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.

The author of MyHDL package has written good blog posts about signal assignments *[7]* and software side of hardware design *[8]*. These ideas are relevant for this chapter.

1.1.2 Objective

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 *[4]*
- 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 to improve the 'two process' model by proposing Object-oriented way for VHDL, lifting all the previously listed drawbacks.

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

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

Listing 1.1: Pipelined multiply-accumulate(MAC) implemented in Pyha

```
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 1.1 shows a MAC component implemented in Pyha (Python to VHDL compiler implemented in the next chapter of this thesis) Operation of this circuit is to multiply the input with coefficient and accumulate the result. It synthesizes to logic shown on Fig. 1.1.

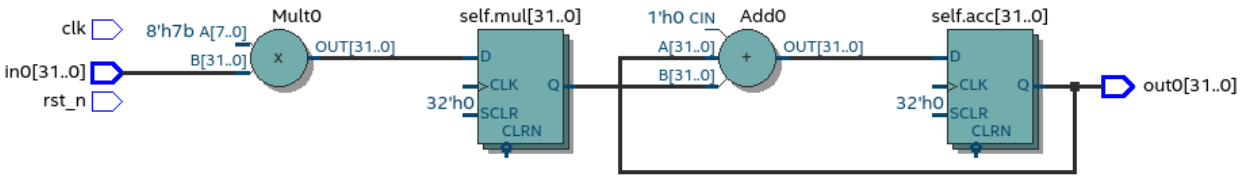

Fig. 1.1: Synthesis result of Listing 1.1 (Intel Quartus RTL viewer)

Main reason to pursue the OOP approach is the modularity and the ease of reuse. Listing 1.2 defines new class, containing two MACs that are to be connected in series. As expected it synthesizes to a series structure (Fig. 1.2).

Listing 1.2: Two MAC's connected in series

```
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
```

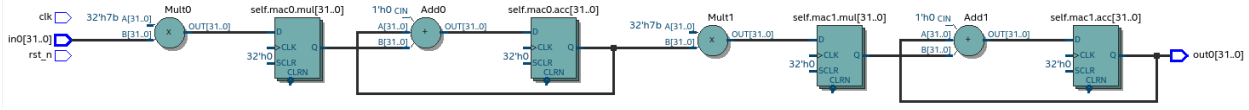

Fig. 1.2: Synthesis result of Listing 1.2 (Intel Quartus RTL viewer)

With slight modification to the 'main' function (Listing 1.3), two MAC's can be connected in a way that synthesizes to parallel structure (Fig. 1.3).

Listing 1.3: Two MAC's in parallel

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

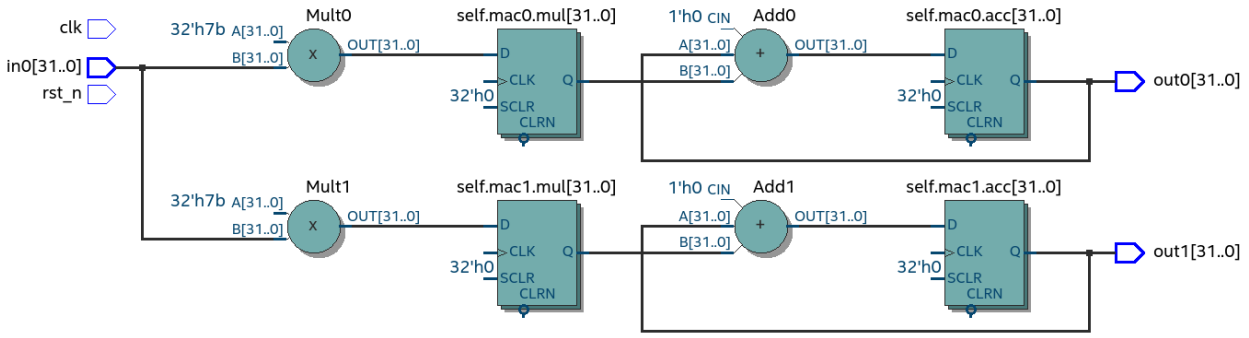

Fig. 1.3: Synthesis result of Listing 1.3 (Intel Quartus RTL viewer)

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

1.1.3 Using SystemVerilog instead of VHDL

SystemVerilog (SV) is the new standard for Verilog language, it adds significant amount of new features to the language *[9]*. 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.

However in my opinion, SV is 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 *[10]*. 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.

Only motivation for using SystemVerilog over VHDL is tool support. For example Yosys *[11]*, open-source synthesys tool, supports only Verilog, however to my knowledge it does not yet support SystemVerilog features. There have been also some efforts in adding VHDL frontend *[12]*.

1.2 Object-oriented style in VHDL

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

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 an 'class' like structure called 'protected types' *[13]*, unfortunately these are not meant for synthesis. Even so, OOP style can be imitated, by combining data in records and passing it as a parameter to 'class functions'. This is essentially the same way how C programmers do it.

Listing 1.4: MAC data model in 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 1.4). 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 1.5: OOP style function in VHDL (implementing MAC)

```
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;
```

OOP style function can be constructed by adding first argument, that points to the data model object (Listing 1.5). 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. Advantage of this is that the procedure may 'output/return' multiple values, as can Python functions.

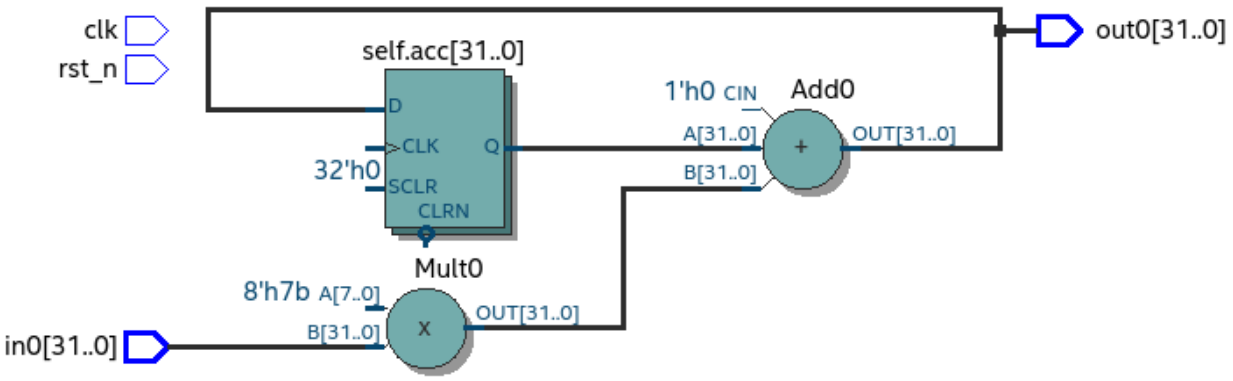

Fig. 1.4: Synthesis result of Listing 1.5 (Intel Quartus RTL viewer)

Synthesis results (Fig. 1.4) show that functionally correct MAC has been implemented. However,in terms of hardware, it is not quite what was wanted. 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 useless.

1.2.1 Understanding registers

Clearly the way of defining registers is not working properly. 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. 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, value assigned to them does not take effect immediately, rather on the next clock edge. That is the basic idea of registers, they take new value on clock edge. When 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 good article about the necessity of signal assignment semantics *[7]*.

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

1.2.2 Inferring registers with variables

While 'signals' and 'signal assignment' are the VHDL way of defining registers, it poses 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 best VHDL books, suggests to use this style in defining sequential logic in VHDL *[14]*. Same semantics are also used in MyHDL *[7]*.

Adapting this style for the OOP data model is shown on Listing 1.6.

Listing 1.6: Data model with **next**

```
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;
```

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 1.7. Only changes are that all the 'register writes' go to the 'nexts' object.

Listing 1.7: Main function using 'nexts'

```
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;
```

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 1.8 defines new function 'update_registers', taking care of this task.

Listing 1.8: Function to update registers

```
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 subset of registers at different clock edge.

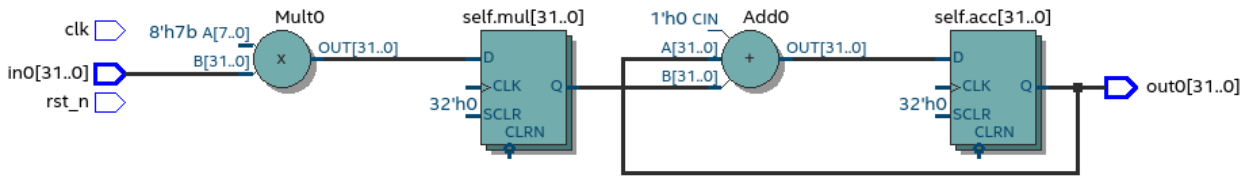

Fig. 1.5: Synthesis result of the upgraded code (Intel Quartus RTL viewer)

Fig. 1.5 shows the synthesis result of the latest code. It is clear that this is now equal to the system presented at the start of this chapter.

1.2.3 Creating instances

General approach of creating instances is to define new variables of the 'self_t' type, Listing 1.9 gives an example of this.

Listing 1.9: Class instances by defining records

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

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));`

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 1.10 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 1.10: Reset function for MAC

```
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 new reset function for each instance.

This can be solved by using VHDL 'generic packages' and 'package instantiation declaration' semantics *[13]*. Package in VHDL just groups common declarations to one namespace.

In case of the MAC class, 'coef' reset value could be set as package generic. Then each new package initialization could define new reset value for it (Listing 1.11).

Listing 1.11: 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. 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 work load.

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

1.2.4 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. Package in VHDL must contain an declaration and body (same concept as header and source files in C).

Listing 1.12 shows the template package for VHDL 'class'. All the class functionality is now in common namespace.

Listing 1.12: 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;
```

1.3 Examples

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

1.3.1 Instances in series

Creating 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 1.13).

Listing 1.13: Datamodel of 'series' class

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

    nexts: next_t;
end record;
```

Next step is to call MAC_0 operation on the input and then pass the output trough MAC_1, whom output is the final output (Listing 1.14).

Listing 1.14: Function that connects two MAC's in series

```
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;
```

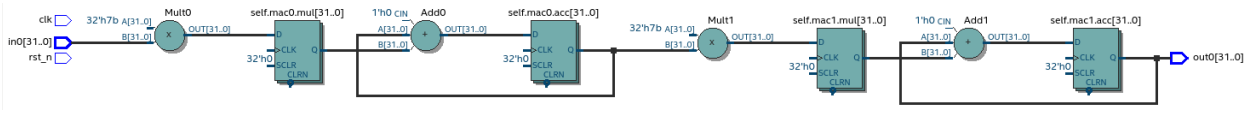

Fig. 1.6: Synthesis result of the new class (Intel Quartus RTL viewer)

Logic is synthesized in series (Fig. 1.6). That is exactly what was specified.

1.3.2 Instances in parallel

Connecting two MAC's in parallel can be done by just returning output of MAC_0 and MAC_1 (Listing 1.15).

Listing 1.15: Main function for parallel instances

```
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;
```

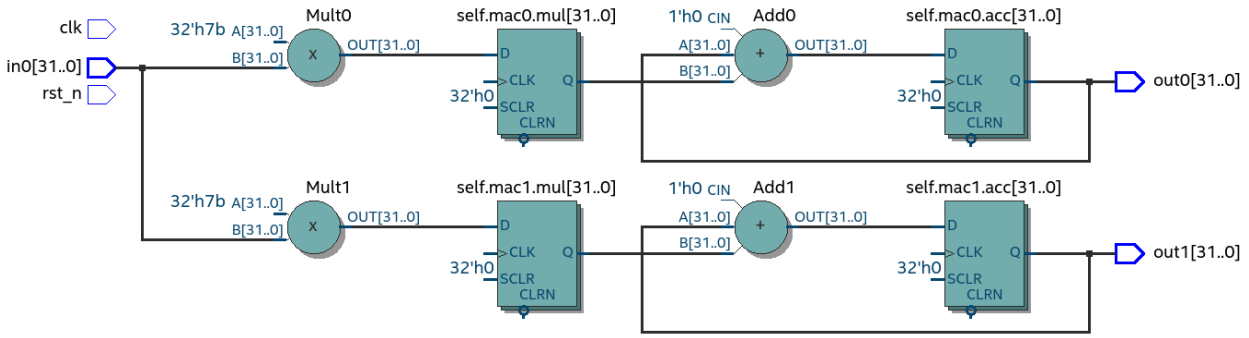

Fig. 1.7: Synthesis result of Listing 1.15 (Intel Quartus RTL viewer)

Two MAC's are synthesized in parallel, as shown on Fig. 1.7.

1.3.3 Parallel instances in different clock domains

Multiple clock domains can be easily supported by updating registers at specified clock domain. Listing 1.16 shows the contents of 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 1.16: Top-level for multiple clocks

```
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;
```

Synthesis result (Fig. 1.8) is as expected, MAC's are still in parallel but now the registers are clocked by different clocks. Reset signal is common for the whole design.

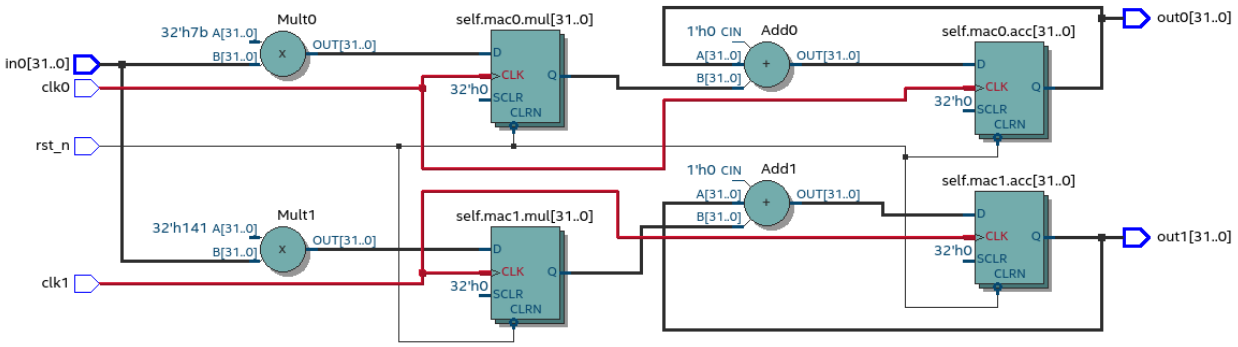

Fig. 1.8: Synthesis result with modified top-level process (Intel Quartus RTL viewer)

Todo

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

1.4 Conclusion

This chapter developed, synthesizable, object-oriented model for VHDL.

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. New programmers can learn this way much faster as the previous knowledge transfers.

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

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.

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 itself may contain arrays of objects.

Chapter 2

Python bindings and simulator

Kohe alguses avada MAC näitega? Kogu thesis põhineb MAC näitel?

Note: No need to go too detailed here!

Why Python? Easy to hack.

Last chapter developed OOP VHDL way. This chapter build on top of this and develops Python bindings. Developing in Python has multiple advantages

- Rich librarys
- Simpler syntax
- Free development tools

Moreover, simulator is provided that so Python designs can be simulated before conversion. Fixed point support is added and described.

Last chapter shows how Python ecosystem can be used to greatly simply the testing/verifying of systems. Model based design.

2.1 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.

2.1.1 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 2.1: 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 2.2: Python variable action

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

Listing 2.1 and Listing 2.2 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 2.3: 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 2.4: Class variable type

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

Listing 2.4 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 *[2]*, 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. *[2]*

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 2.5: 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 2.6: Type problems

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

2.1.2 Conversion methodology

Methodology is RedBaron.

VHDL is known as a strongly typed language in addition to that it is very verbose. Python is dynamically typed and is basically as least verbose as possible.

Based on the results of previous chapter it is clear that specific Python code can be converted to VHDL. Doing so requires some way of parsing the Python code and outputting VHDL.

In general this step involves using an abstract syntax tree (AST). MyHDL is using this solution.

However RedBaron offers a better solution. RedBaron is an Python library with an aim to significally simply operations with source code parsing. Also it is not based on the AST, but on FST, that is full syntax tree keeping all the comments and stuff.

Here is a simple example:

```
>>> red = RedBaron('a = b')
>>> red
0   a = b
```

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

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

Now Pyha defined a mirror node for each of RedBaron nodes, with the goal of turning the code into VHDL. For example in the above example main node is AssignmentNode, this could be modified to change the '=' into ':=' and add ';' to the end of line. Resulting in a VHDL compatible statement:

```
a := b;
```

2.1.3 Basic conversions

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

2.1.4 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 2.7: 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);
```

2.1.5 Converting classes

Extracting the data model

Instances

Overall converting classes is simple as they consist of functions.

2.1.6 Types

This chapter gives overview of types supported by Pyha.

Integers

Integer types and operations are supported for FPGA conversion with a couple of limitations. First of all, Python integers have unlimited precision *[3]*. This requirement is impossible to meet and because of this converted integers are assumed to be 32 bits wide.

Conversion wize, all inger objectsa are mapped to VHDL type 'integer', that implements 32 bit signed integer. In case integer object is returned to top-module, it is converted to 'std_logic_vector(31 downto 0)'.

Booleans

Booleans in Python are truth values that can either be True or False. Booleans are fully supported for conversion. In VHDL type 'boolean' is used. In case of top-module, it is converted to 'std_logic' type.

Floats

Floating point values can be synthesized as constants only if they find a way to become fixed_point type. Generally Pyha does not support converting floating point values, however this could be useful because floating point values can very much be used in RTL simulation, it could be used to verify design before fixed point conversion.

Floats can be used as constants only, in coperation with Fixed point class.

2.1.7 User defined types / Submodules

Support for VHDl conversion is straightforward, as Pyha modules are converted into VHDL struct. So having a submodule means just having a struct member of that module.

2.1.8 Lists

All the previously mentioned convertible types can be also used in a list form. Matching term in VHDL vocabulary is array. The difference is that Python lists dont have a size limit, while VHDL arrays must be always constrained. This is actually not a big problem as the final list size is already known.

VHDL being an very strictly typed language requires an definition of each array type.

For example writing `l = [1, 2]` in Python would trigger the code shown in `vhdl-int-arr`, where line 1 is a new array type definitiaon and a second line defines a variable `a` of this type. Note that the elements type is deduced from the type of first element in Python array the size of defined array is as `len(l)-1`.

Listing 2.8: VHDL conversion for integer array

```
1 type integer_list_t is array (natural range <>) of integer;
2 l: integer_list_t(0 to 1);
```

Constants? Interfaces?

2.2 Python model and Simulation

This chapter introduces the way of writing hardware designs in Python. Simulator info is provided also. This chapter does not worry about conversion process.

2.2.1 Object-orientation in Python

Unit object is Python class as shown on

Listing 2.9: Basic Pyha unit

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

    def main(self, input):
        pass
```

Listing 2.9 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.

As for the VHDL model, we can assume that all the variables in the 'self' scope are registers.

2.2.2 Writing hardware in Python

As shown in previous chapter, traditional language features can be used to infer hardware components. One must still keep in mind of how the code will convert to hardware. For example all loops (For) will be unrolled, this dentotes that the loop control must have finitive limit.

Another point to note is that every arithmetical operator used will use up resorce. There is a big difference between hardware and software programming, using operators in software takes up time but in hardware they will all run in parallel so no addtional time is used BUT resource. There are ways to share the operators to trade resource for time.

One thing that is not natively supported in python is registers, for this we did special stuff in VHDL section, basically the same can be done in Python domain.

statemachines?

2.2.3 Adding registers support

The init function is used to determine the startup valu

Working with registers is implemented in a same way as in VHDL model. Meaning there are buffered. For this there is metaclass action, that allows chaning the process of class creation. Metaclass copies all the object data model to a new variable called 'next'. Thus automating the creation of the buffer values.

How signal assignments can work in Python.

Moreover, automatically function is created for updating the registers, it was named 'update registers' in VHDL model, now it is named '_pyha_update_self'. The effect of it is exactly the same, it copies 'next' variables to 'current', thus mimicing the register progress.

2.2.4 Reset values

In hardware is is important to be able to set the reset/power on values for the registers. In same sense this is important for class instance creation.

Listing 2.10: Reset example

```
class SimpleClass(HW):
    def __init__(self):
        self.reg0 = 123
        self.reg1 = 321
```

Listing 2.10 shows an example class, that defines two registers. Initial values for them will be also their hardware reset values.

2.2.5 State-machines

2.2.6 Simulation

Simulation of single clock designs is trivial. Main function must be called and then '_pyha_update_self'. This basically is an action for one clock edge.

Here is an example that pushes some data twough the MAC component. This simulation result is equal to the GHDL simulation and generated netlist GATE simulation.

Todo

add fixed point type here? rather keep separte? Convertable subset?

Last chapter shows how to further improve the simulation process by using helper function provided by Pyha.

2.2.7 Conclusions

Pyha extends Python language to add support for hardware also simulation is possible.

2.3 Testing, debugging and verification

This chapter aims to investigate how modern software development techniques coulde be used in design of hardware.

While MyHDL brings development to the Python world, it still requires the make of test-benches and stuff. Pyha aimst to simplify this by providing higl level simulation functions.

2.3.1 Background

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.

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.

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)

2.3.2 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.

2.3.3 Ipython notebook

Simple example of docu + test combo. It is interactive environment for python. Show how this can be used.

2.4 Conclusions

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

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

Bibliography

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

[2] 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-/.

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

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

[5] 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.

[6] 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.

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

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

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

[10] 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/.

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

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

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

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