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

[Thesis code]
Gaspar Karm

VHDL as Object-oriented Intermediate Language and Python bindings

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

1.1 Objective

The most commonly used design 'style' for synthesisable 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 implemenet the desired function- ality. 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. It is not uncommon that to extract the functionality of dataflow code, a block diagram has to be drawn to indentify the dataflow and depend- ecies between the statements. The readability of dataflow VHDL code can compared to an ordinary schematic where the wires connecting the various blocks have been removed, and the block inputs and outputs are just labeled with signal names! *[1]*

A problem with the dataflow method is also the low abstraction level. The functionality is coded with simple constructs typically consisting of multiplexers, bit-wise operators and conditional assignments (if-then-else). The overall algorithm might be very difficult to recognize and debug *[1]*.

The biggest difference between a program in VHDL and standard programming lan- guage such C, is that VHDL allows concurrent statements and processes that are sched- uled for execution by events rather than in then order they are written. This reflects indeed the dataflow behaviour of real hardware, but becomes difficult to understand and analyse when the number of concurrent statments passes some threashold (e.g. 50). On the other hand, analysing the behaviour of programs witten in sequential program- ming languages does not become a problem even if the program tends to grow, since there is only one thread of control and execution is done sequentially from top to bot- tom *[1]*.

In order to improve readability and provide a uniform way of encode the algorithm of a VHDL entity, the two-process method only uses two processes per entity: one process that contains all combinational (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 *[1]*.

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

- It is applicable to single-clock designs. While it is true that majority of the designs are single clock, it is significant limitation anyways.
- The 'structured' part can be only used to define combinatory logic, registers must be still inferred by signals assigments.
- It still relies on many of the VHDL data-flow features, for example design reuse is arcieved trough the use of entityes and port maps.

The goal of this section is improve the 'two process' model by proposing Object-oriented way for VHDL. It gets rid of previous drawbacks.

Gaols is to introduce alternative model, where same things can be achieved but with programming model much closer to everyday programmers.

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 section.

Siin parem sissejuhatus, miks see Pythoni junn siin?

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 some coefficient and then accumulate the result. It synthesizes to logic shown on Fig. 1.1.

This chapter tries to find and VHDL model that could easly accomodate this OOP based

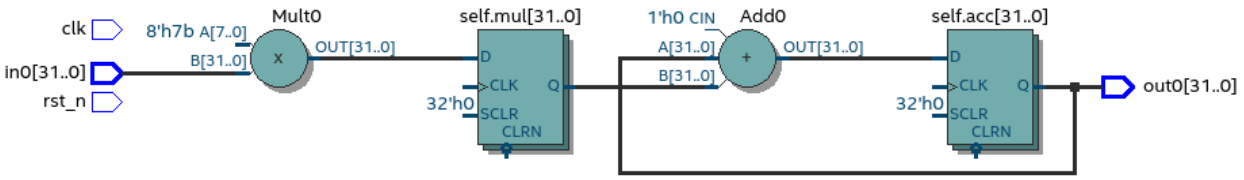

Fig. 1.1: Synthesis result of Listing 1.1 (Intel Quartus RTL viewer)

style.

One problem in VHDL is that reusing components is not trivial, programmers must do 'wiring' work that is error prone, declaring arrays of components is even harder. The main reason to pursue the OOP approach is the modularity and the ease of reuse.

On the other hand these operations are easy with OOP approach, for example Listing 1.2 defines new class, that has two MACs in series. As expected it synthesizes to a structure where two MACs are connected in series, shown on 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
```

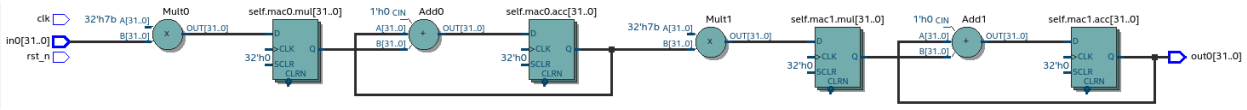

Fig. 1.2: Synthesis result of Listing 1.2 (Intel Quartus RTL viewer)

With slight modification to the 'main' function (Listing 1.3), two MAC's can be connected in parallel instead. As expected this would synthesize to parallel MAC's as shown on 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
```

It is clear that OOP style could significantly simplify the design of hardware.

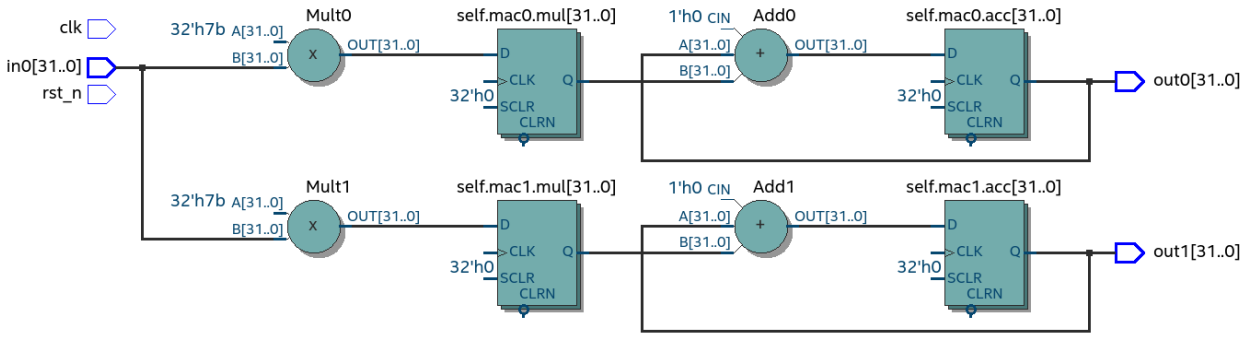

Fig. 1.3: Synthesis result of Listing 1.3 (Intel Quartus RTL viewer)

Basically in this chapter we are looking to develop an VHDL model that could easly describe these previously listed examples.

Major features that we are looking for:

- OOP style for conversion ease
- Familiarity to normal programmers
- Must be fully synthesisable
- Should not limit the hardware description stuff, like multiple clocks
- Unify/simplify Python to VHDL conversion

1.2 Background

There have been previous study regarding OOP in VHDL. In *[2]* proposal was made to extend VHDL language with OOP semantics, this effort ended with development of OO-VHDL *[3]*, that is VHDL preprocessor that could turn proposend extensions to standard VHDL. This work was done in ˜2000, current status is unknown, it certanly did not make it to the VHDL standard.

While the *[3]* tried to extend VHDLs data-flow side of OOP, there actually exsists another way to do it, that is inherited from ADA.

There are many tools on the market that convert some higher level language to VHDL, for example MyHDL converts Python to VHDL and Verilog. However these tools only make use of the very basic elements of VHDL language. The result of this is that coneversion process is complex and hard to understand. Also the output VHDL generally does not keep design hirarchy and is very hard to read for humans.

While other HDL converters use VHDL/Verilog as low level conversion target. Pyha goes other way around, as shown by the Gardner study *[1]*, VHDL language can be used with quite high level progrmaming constructs. Pyha tries to take advantage of this.

The author of MyHDL package has written some good blog posts about signal assigmennts and software side of hardware design *[4]*, *[5]*. These ideas are relaveant for this chapter.

Jiri Gaisler has proposed an 'Structured VHDL design method' in the ~2000 *[1]*. He proposes to raise the hardware design abstraction level by instead of writing 'dataflow' style. Use two process method where the algorithmic part is described by the regular function in one process and registers are in another process.

Gaisler notest that functions only good for combinatory logic and in one clock domian, try to improve that.

The goal of the two-process method is to:

- Provide uniform algorithm encoding
- Increase abstraction level
- Improve readability
- Clearly identify sequential logic
- Simplify debugging
- Improve simulation speed
- Provide one model for both synthesis and simulation

This work improves upon the work of Jiri Gaisler.

Siin v]ib ka kirjutada VHDL vs Verilog asjadest, Verilog populaarsem? OS tools.

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. *[6]*. Most of the added synthesizable features already existed in VHDL, making the syhtnesizable 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, that 'feature' is actually written into the language reference manual *[7]*. VHDL would error out the simulation.

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 mainly as IR language.

Only motivation for using SystemVerilog over VHDL is tool support. For example Yosys *[8]*, 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 to Yosys *[9]*.

1.3 Object-oriented style in VHDL

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

Basic idea of OOP is to bundle up some common data and define functions that can perform actions on this data. Then one could define multiple sets of that data. This idea could fit well with hardware design, we could define 'data' as registers and functions as logic between registers (combinatory logic).

VHDL includes an 'class' like structure called 'protected types' *[10]*, 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 record as 'self', to make it 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 argumnet, 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.

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 on wrong location.

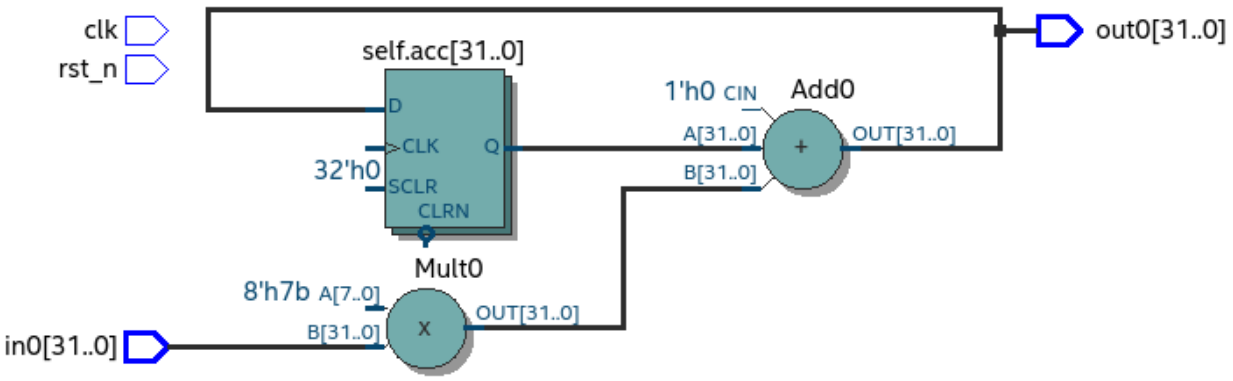

Fig. 1.4: Synthesis result of Listing 1.5 (Intel Quartus RTL viewer)

In fact, the signal path from **in0** to **out0** contains no registers at all, making this design rather useless.

1.3.1 Understanding registers

Clearly the way of defining registers is not working properly. Problem is that we expected the registers to work in the same way as 'class variables' in traditional programming languages, but registers work a bit differently.

In traditional programming, class variables are very similar to function local variables. The difference is that class variables can 'remember' the value and have bigger scope. Local variables exsist 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 the new value on clock edge, and when we set the value **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, we could say that the assignment is delayed by one function call.

VHDL defines an special type of objects, called signals, for these kind of variables. VHDL defines a special assignment operator for this kind of delayed stuff, it is called 'signal assignment'. It is defined like `a <= b`.

Jan Decaluwe, the author of MyHDL package, has written good 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.

1.3.2 Inferring registers with variables

While 'signals' and 'signal assignment' is the VHDL way of defining registers, it poses a major problem because they are hard to map to any other language than VHDL, making conversion hard. 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 *[11]*. Same semantics are also used in MyHDL *[4]*.

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, but extends the structure to include the 'nexts' object that can be used to assign new value for the register, 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.

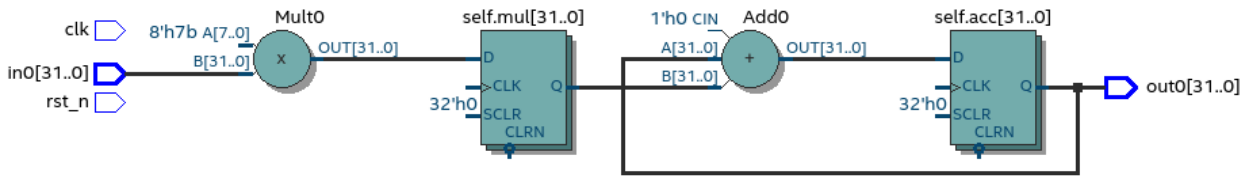

Fig. 1.5: Synthesis result of the upgraded code (Intel Quartus RTL viewer)

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.

1.3.3 Creating instances

General approach of creating instances is to define new variables of the 'self_t' type, [vhdl-instance] gives an example of this.

Listing 1.9: Class instances by defining records

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

Creating new instances of the VHDL package can be done by using 'package instantiation declaration' and 'package generics' *[10]*. In case of the MAC class, we would like to set the 'coef' value for new instances. This is archived by defining the package with a 'generic' definition and initialize new packages like shown on `vhdl-package-init`. In the reset function we could then use the generic 'COEF' for 'coef' init value.

Listing 1.10: 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.

Listing 1.11: Initialize new package MAC_0, with 'coef' 123

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

Note that problem is with the reset values. In case default reset values are required,

There are two ways around this issue:

- Instead of using reset function, reset registers with assignment
- For each instance create only new reset function
- For each instance manually create new package with modified reset function

The first option proposes setting reset values inline on reset, for example, `self: self_t := (mul=>0, acc=>0, coef=>123, nexts=>(mul=>0, acc=>0, coef=>123));`. Problem with this method is that it needs to set all the members of struct (including 'nexts'), this will get unmaintainable very quiqly, imagine having an instance that contains another instance or even array of instances.

Second option would keep one package for each objects but different reset functions. This may end up in error-prone code where wrong reset function is used accitendally.

Last option proposes to manually do the work of package initialization, that is for each instance make a new package. This will end up in alot of duplicated code.

In general all of these solutions have problems, in this work i have chosen the last option, because it is safe unlike the second option. In the end creating of new packages is automated by the Python bindings developed in the next chapter.

1.3.4 Initial register values

The OOP model for VHDL is almost complete, only thing it misses is initialization of registers. In traditional programming languages this is done by the class constructor, executing automatically for new objects.

In the sense of hardware, this operation can be called 'reset'. Listing 1.12 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 1.12: 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;
```

1.3.5 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 1.13 shows the template package for VHDL 'class'. All the common functionality is now grouped into package.

Listing 1.13: 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;
```

1.3.6 Examples

This chapter provides examples that make use of the MAC model and OOP.

Instances in series

This paragraph shows how to create a new class that itself includes two MAC's connected in series, that is, signal flows is as **in** -> MAC0 -> MAC1 -> **out**.

Assuming we have already created two MAC packages called MAC_0 and MAC_1, connecting these in series is simple.

Listing 1.14: Datamodel and main function of 'series' class

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

    nexts: next_t;
end record;

procedure main(self:inout self_t; a: integer; ret_0:out integer) is
    variable out_tmp: integer;
begin
    MAC_0.main(self.mac0, a, ret_0=>out_tmp);
```

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

Listing 1.14 shows the important parts of the series MAC implementation. Datamodel consists of two MAC objects and the main function just calls the main of these objects. Output of MAC_0 is fed into MAC_1, which results in final output.

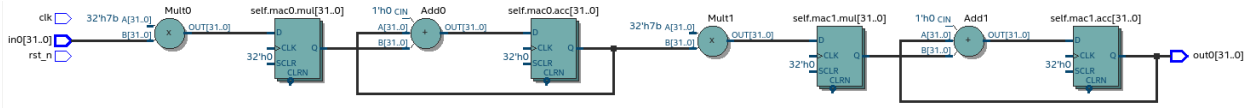

Fig. 1.6: Synthesis result of Listing 1.14 (Intel Quartus RTL viewer)

Logic is synthesized in series, as shown on Fig. 1.6. That is exactly what was specified.

Instances in parallel

Connecting two MAC's in parallel can be done by just adding one output for the main function and returning output of MAC_0 as separate output instead of input to MAC_1, this is shown on 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;
```

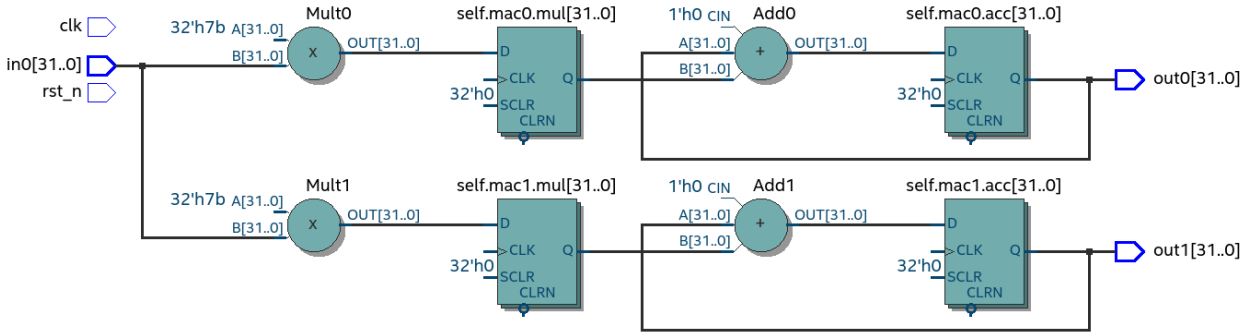

Fig. 1.7: Synthesis result of Listing 1.15 (Intel Quartus RTL viewer)

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

Parallel instances in different clock domains

Multiple clock domains can be easly supported by just updating registers at specified clock edges. Listing 1.16 shows the contents of top-level process, where we intend to have 'clk0' for 'mac0' and 'clk1' for 'mac1'. Beauty of this method is that nothing has to be changed in the 'main' functions.

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;
```

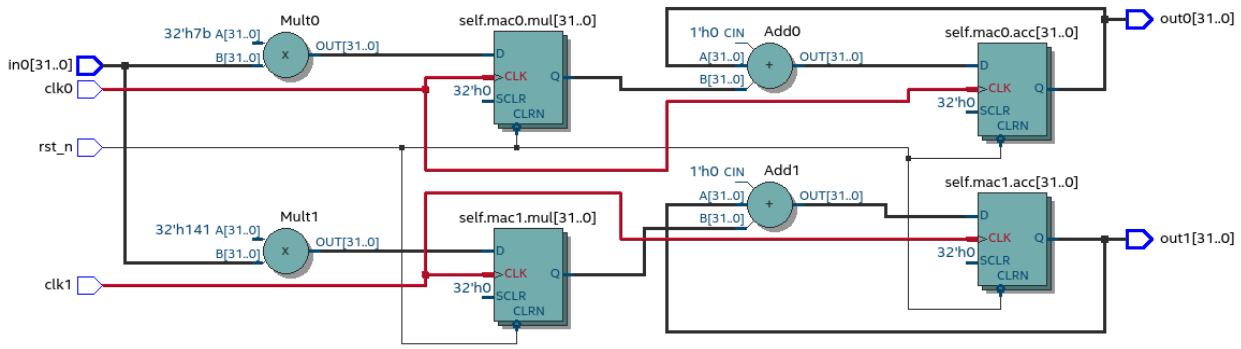

Fig. 1.8: Synthesis result with modified top-level process (Intel Quartus RTL viewer)

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.

Mention Qsys and interconnects here?

1.4 Conclusion

This work started from the Gaisler study, while he presented two process design methology, hes use of functions was limited to de combinatory logic only and overall was limited to single clock. He was also using many of the awkward VHDl features.

This work extends the gaisler stuff by proposing OOP model into VHDL and introducing the way of defining registers using only registers, this allows the functions to work with registers aswell. In addtion, one clock domain restriction is lifted.

Major advantage of this model is that it does not use any specialized data-flow features of VHDL (except top level entity). New programmers can learn this way of programming much quicker as mostly they can make use of the stuff they already know. Only some rules like that stuff must be assigned to 'next' must be known.

Another benefit of OOP style model is that is significantly simplifies converting other OOP languages to VHDL and that was the major goal of this section. Next section shows how and experimental Python compiler is built on top of this.

Every register of the model is kept in record, it is easy to create shadow registers for the whole module. Everything is concurrent, can debug and understand.

Easier to understand for new programmers, this model contains only elements that should be already familiar for programmers dealing with normal languages.

As demonstrated, proposed model is synthesizable with Intel Quartus toolset. This model has also been used in bigger designs, like frequency-shift-keying receiver implemented on Altera Cyclone IV device. There has been no problems with hierarchy depth, that is objects can contain objects which itself may contain arrays of objects and so on.

Bibliography

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

[2] 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.

[3] 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.

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

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

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

[7] 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/.

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

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

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

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