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

Hardware design with Pyha

This chapter introduces the main contribution of this thesis, Pyha - tool to design digital hardware in Python.

Pyha proposes to program hardware in same way as software is programmed, much of this chapter is focused on showing differences between hardware and software constructs.

First half of this chapter is focuses on the basic hardware construct and how they can be described using Pyha. Examples in this section operate on the integer data type, to reduce the initial complexity.

Second half of this chapter introduces the fixed-point type and demonstrates how it can be used to develop moving average and linear phase DC removal filters.

All the examples presented in this chapter can be found online HERE, these include all the Python sources, unit-tests, VHDL conversion files and Quartus project for synthesis.

Todo

organise examples to web and put link

1.1 Introduction

Pyha follows the object-oriented design paradigm, basic design unit is an Python class, that is derived from HW subclass (to inherit hardware related functionality).

Model based design is encouraged, where model is non-synthesisable code for simplest possible implementation. Most often the model is implemented with a call to Numpy or Scipy (Python scientific computing libraries). Model helps the testing process and can also serve as an documentation.

Listing 1.1 shows the implementation of simple adder circuit. In Pyha all class variables are interpreted as hardware registers. The `__init__` function may contain any Python code to evaluate reset values for registers. The `model_main` function is reserved for defining the model and `main` as the top level for synthesis. Note that the `model_main` is completely ignored for synthesis.

Listing 1.1: Simple adder, implemented in Pyha

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

    def main(self, x):
        y = x + self.coef
        return y

    def model_main(self, xl):
        # for each element in xl add 1
        yl = [x + self.coef for x in xl]
        return yl
```

Notice how the `model_main` function works on lists, it gets all the inputs at once, this enabled vectorized implementations. The `main` however works on single input, as is the hardware way.

Note: All the examples in this chapter include the model implementation. In order to keep code examples smaller, future listings omit the model code.

1.1.1 Simulation and testing

Pyha designs can be simulated in Python or VHDL domain. In addition, Pyha has integration to Intel Quartus software, it supports running GATE level simulations (simulation of synthesized logic).

Pyha provides functions to automatically run all the simulations on the set of input data. Listing 1.2 shows an example unit test for the 'adder' module.

Listing 1.2: Adder tester

```
x =      [1, 2, 2, 3, 3, 1, 1]
expect = [2, 3, 3, 4, 4, 2, 2]

dut = Adder(coef=1)
assert_simulation(dut, expect, x)
```

The `assert_simulation(dut, expect, x)` runs all the simulations (Model, Pyha, RTL and GATE) and asserts the results equal the `expexct`.

In addition, `simulations(dut, x)` returns all the outputs of different simulations, this can be used to plot the results, as shown on Fig. 1.1.

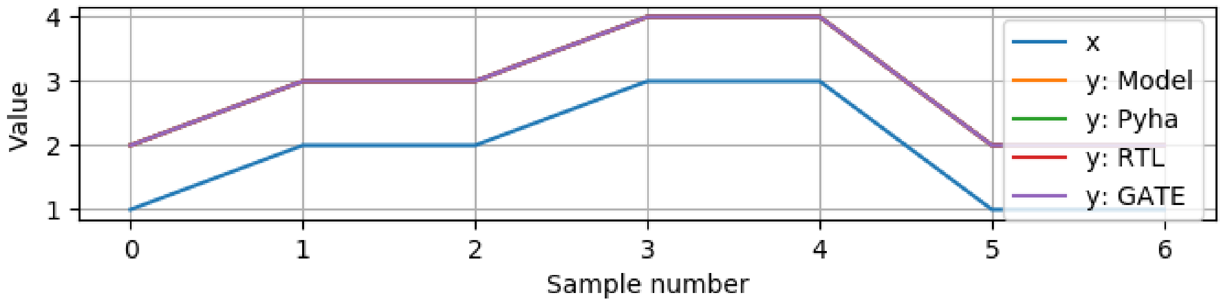


Fig. 1.1: Simulation input and outputs

More information about the simulation functions can be found in the APPENDIX.

Todo

Add simulation function definitins to appendix.

1.1.2 Synthesis

Synthesis is required to run the GATE level simulations, Pyha integrates to the Intel Quartus software in order to archive this.

The synthesis target device is EP4CE40F23C8N, of Cyclone IV family. This is the same FPGA that powers the latest LimeSDR chip and the BladeRF board. In general it is a low cost FPGA with following features *[13]*:

- 39,600 logic elements
- 1,134Kbits embedded memory
- 116 embedded 18x18 multipliers
- 4 PLLs
- 200 MHz maximum clock speed

One useful tool in Quartus software is RTL viewer, it can be opened via `Tools->Netlist viewers->RTL viewer`. RTL viewer visualizes the synthesised hardware for the Pyha design, this chapter uses it extensively.

Fig. 1.2 shows the RTL of the adder circuit. Notice that the integer types were synthesised to 32 bit logic ([31..0] is the signal width).

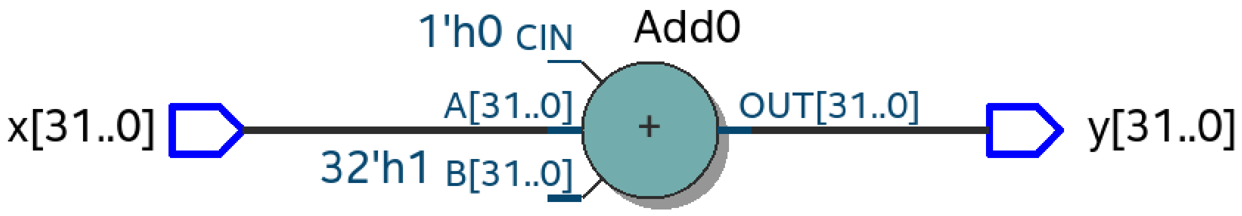


Fig. 1.2: RTL of the adder circuit

1.1.3 Design flow

Suggested design flow for Pyha designs is model based development with test-driven approach. Meaning that the unit tests should be developed to assert the performance of the model. For unit tests use the Pyha `simulate` functions, so that the same tests can be later executed for hardware models.

Last step is to implement the synthesizable code (`main`), development is greatly simplified if enough unit tests were collected while developing the model.

Todo

Needs more info, make figure, fixed point?

Note: Following examples in this chapter tend to ignore the model and unit-testing part and rush to the hardware implementation, since this is the focus of this chapter.

1.2 Stateless designs

Designs that don't contain any memory elements can be considered stateless. This is also called combinatory logic in hardware terms. In software world, this can be understood as an function that only uses local variables, using class variables would introduce state.

1.2.1 Basic operations

Listing 1.3 shows the Pyha design, featuring circuit with one input and two outputs. Note that the `b` output is dependent of `a`.

Listing 1.3: Basic stateless design

```
class Basic(HW):
    def main(self, x):
        a = x + 1 + 3
        b = a * 314
        return a, b
```

Fig. 1.3 shows that each add instruction is synthesised to an actual resource in the FPGA fabric. The a output is formed by running the x signal trough two adders (one adding 1 and next 3). The b has extra multiplier on signal path.

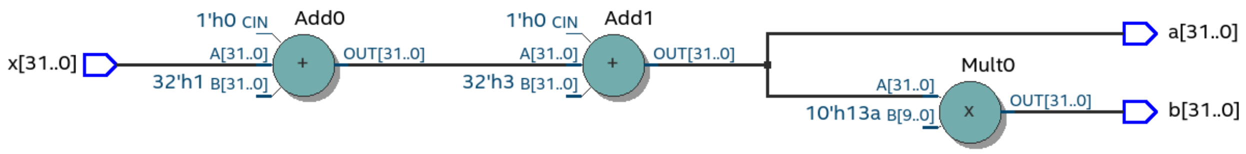
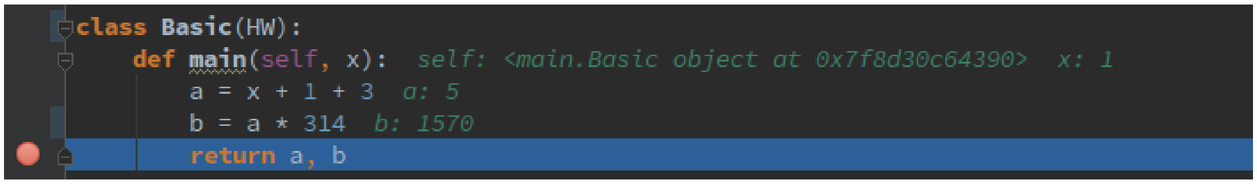


Fig. 1.3: Synthesis result of Listing 1.3 (Intel Quartus RTL viewer)

This example shows that in hardware operations have a price in terms of resource usage. This is a big difference to software, where operations cost execution time instead.

Key idea to understand is that while the software and hardware execute the main function in different ways, they result in same output, so in that sense they are equal. This idea is confirmed by Pyha simulation, reporting equal outputs for all simulations.

Huge upside of Pyha is that designs can be debugged. Pyha simulations just runs the main function so all kinds of Python tools can be used. Fig. 1.4 shows a debugging session on the Listing 1.3 code. Using Python tools for debugging can greatly increase the designers productivity.



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

Fig. 1.4: Debugging using PyCharm (Python editor)

1.2.2 Conditional statements

Main conditional statement in Python is if, it can be combined with elif and else. All of these are convertible to hardware. Listing 1.4 shows an example of basic if else statement.

Listing 1.4: Select add amount with if

```python
class If(HW):
    def main(self, x, condition):
        if condition == 0:
            y = x + 3
        else:
            y = x + 1
        return y
```

Fig. 1.5 shows that in hardware the `if` clause is implemented by the 'multiplexer' component. It works by, based on condition, routing one of the inputs to the output. For example if `condition == 0` then bottom signal path is routed to output. Interesting thing to note is that both of the adders are 'executing', just one of the result is thrown away.

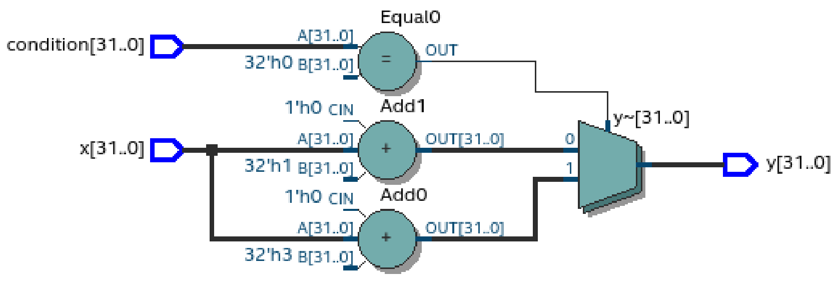


Fig. 1.5: Synthesis result of Listing 1.4 (Intel Quartus RTL viewer)

Once again, all the simulations result in equal outputs.

1.2.3 Loop statements

All the loop statements will be unrolled in hardware, this indicates that the loop control statement cannot be dynamic.

Listing 1.5 shows an simple `for` example, that adds [0, 1, 2, 3] to the input signal.

Listing 1.5: `for` example

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

        return y
```

Fig. 1.6 shows that RTL consists of chained adders, that have been also somewhat optimized.

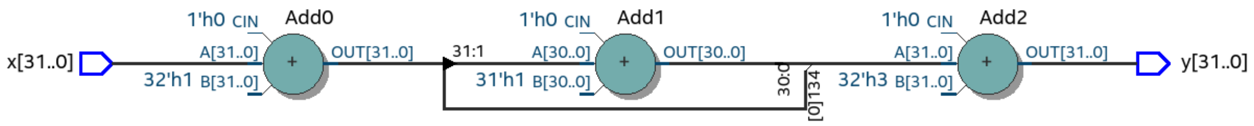


Fig. 1.6: Synthesis result of Listing 1.5 (Intel Quartus RTL viewer)

The RTL may make more sense if we consider the unrolled version, shown on Listing 1.6.

Listing 1.6: Unrolled `for`, equivalent to Listing 1.5

```
y = x
y = y + 0
y = y + 1
y = y + 2
y = y + 3
```

Most importantly, all the simulations provide equal results.

1.2.4 Function calls

So far only the `main` function has been used to define logic. in Pyha `main` function is just the top level function that is first called by simulation and conversion processes. Other functions can freely be defined and called as shown on Listing 1.7.

Listing 1.7: Calling an function in Pyha

```
class Functions(HW):
    def adder(self, x, b):
        y = x + b
        return y

    def main(self, x):
        y = self.adder(x, 1)
        return y
```

The synthesis result of Listing 1.7 is just an adder, there is no indication that a function call has been used, one can assume that all functions are inlined during the synthesis process.

Note that calling the function multiple times would infer parallel hardware.

1.2.5 Conclusions

This chapter has demonstrated that many of the software world constructs can be mapped to the hardware and the outputs of the software and hardware simulations are equal. Some limitations exist, for example the `for` loop must be unrollable in order to use in hardware.

Major point to remember is that every statement converted to hardware costs resources. This is different to the software world where statements instead cost execution time.

1.3 Designs with memory

So far, all the designs presented have been stateless (without memory). Often algorithms need to store some value for later use, this indicates that the design must contain memory elements.

This chapter gives overview of memory based designs in Pyha.

1.3.1 Accumulator and registers

Consider the design of accumulator, it operates by sequentially adding up all the input values. Listing 1.8 shows the Pyha implementation, class scope variable is defined in the `__init__` function to store the accumulator value.

Listing 1.8: Accumulator implemented in Pyha

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

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

Trying to run this, would result in Pyha error, suggesting to change the line 6 to to `self.next.acc = ....` After this code is runnable, reasons for this modification are explained shortly.

Synthesis result shown on the Fig. 1.7 features an new element known as register.

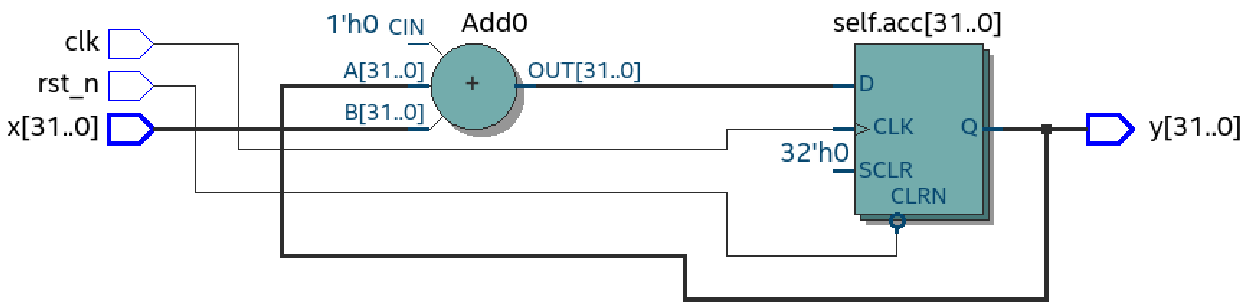


Fig. 1.7: Synthesis result of Listing 1.5 (Intel Quartus RTL viewer)

Register

Todo

this section is not finished

In software programming, class variables are the main method of saving the some information from function call to another.

Register is an hardware memory component, it samples the input signal `D` on the edge of the `CLK` signal. In that sense it acts like a buffer.

One of the new signals on the RTL figure is `clk`, that is a clock signal that instructs the registers to update the saved value (`D`).

In hardware clock is a mean of synchronizing the registers, thus allowing accurate timing analsys that allows placing the components on the FPGA fabric in such way that all the analog transients happen **between** the clock edges, thus the registers are guaranteed to sample the clean and correct signal.

Registers have one difference to software class variables, the value assigned to them does not take effect immediately, but rather on the next clock edge. When the value is set at **this** clock edge, it will be taken on **next** clock edge.

Pyha tries to stay in the software world, so the clock signal can be abstracted away by thinking that it denotes the call to the 'main' function. Meaning that registers update their value on every call to `main` (just before the call).

Think that the `main` function is started with the **current** register values known and the objective of the `main` function is to find the **next** values for the registers.

In DSP systems one important variable is sample rate. In hardware the maximum clock rate and sample rate are basically the same thing. In Digital signal processing applications we have sampling rate, that is basically equal to the clock rate. Think that for each input sample the 'main' function is called, that is for each sample the clock ticks.

Note that the way how the hardware is designed determines the maximum clock rate it can run off. So if we do a bad job we may have to work with low sample rate designs. This is determined by the worst critical path.

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

`rst_n` signal can be used to set initial states for registers, in Pyha the initial value is determined by the value assigned in `__init__`, in this case it is 0.

Testing

Running the same testing code results in a Fig. 1.8. It shows that the **model** simulation differs from the rest of the simulations. It is visible that the hardware related simulations are **delayed by 1**. This is the side-effect of the hardware registers, each register on the signal path adds one sample delay.

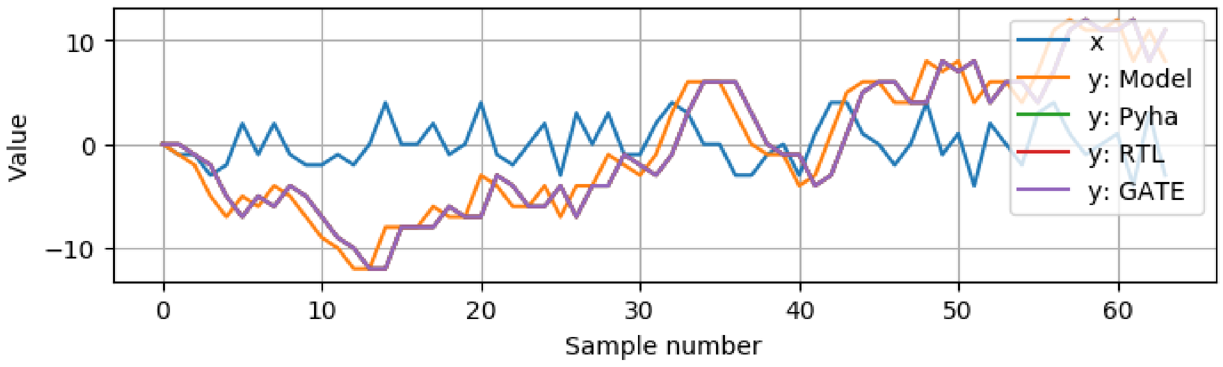


Fig. 1.8: Simulation of the accumulator (x is random integer [-5;5])

Pyha provides an `self._delay` variable, that hardware classes can use to specify their delay. Simulation functions can read this variable and compensate the simulation data so that the delay is compensated, that eases the design of unit-tests.

All the simulations match in output (Fig. 1.9), after setting the `self._delay = 1` in the `__init__`.

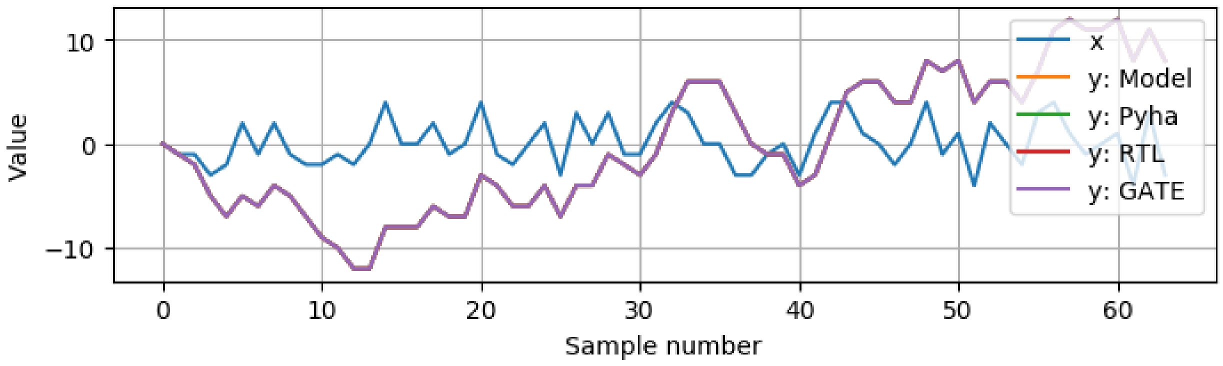


Fig. 1.9: Simulation of the delay **compensated** accumulator (x is random integer [-5;5])

1.3.2 Block processing and sliding adder

One very common task in DSP designs is to calculate results based on some number of input samples (block processing). Currently the `main` function has worked with the single input sample, this can now be changed by keeping the history with registers.

Consider an algorithm that adds the last 4 inputs. Listing 1.9 shows an implementation that keeps track of the last 4 inputs and sums them. Note that the design also uses the output register `y`.

Listing 1.9: Sliding adder algorithm

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

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

        # add all element in the list
        sum = 0
        for a in self.shr:
            sum = sum + a

        self.next.y = sum
        return self.y
```

The `self.next.shr = [x] + self.shr[:-1]` line is also known as an 'shift register', because on every call shifts the list contents right and adds new `x` as the first element. Sometimes the same structure is used as an delay-chain, because the sample `x` takes 4 updates to travel from `shr[0]` to `shr[3]`. This is a very common element in hardware DSP designs.

Fig. 1.10 shows the RTL for this design, as expected the `for`

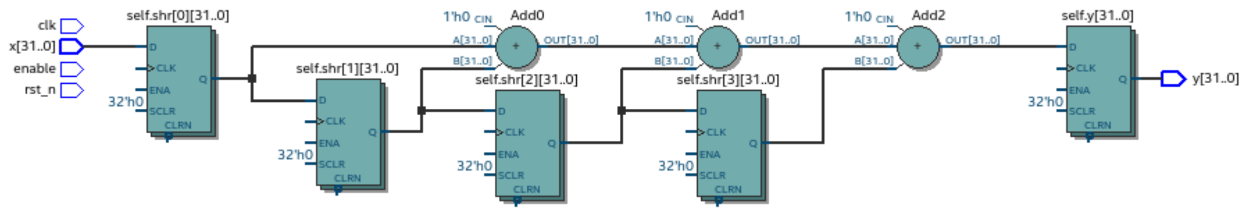


Fig. 1.10: Synthesis result of Listing 1.9 (Intel Quartus RTL viewer)

Optimizing the design

This desing can be made generic by chaning the `__init__` function to take the window length as a parameter (Listing 1.10).

Listing 1.10: Generic sliding adder

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

Problem with this design is that it starts using more resources as the `window_len` gets larger as every stage requires an separate adder. Another problem is that the critical path gets longer decreasing the clock rate. For example, the design with `window_len=4` synthesises to maximum clock of 170 MHz, while `window_len=6` to only 120 MHz.

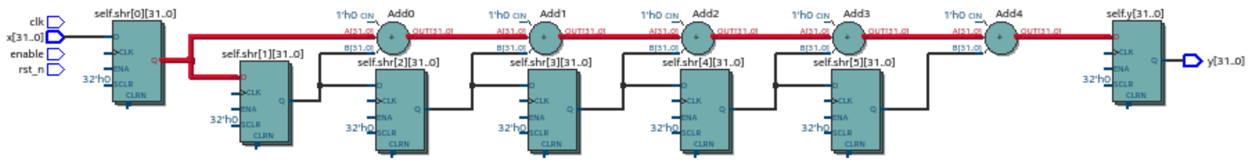


Fig. 1.11: RTL of `window_len=6`, red line is critical path (Intel Quartus RTL viewer)

In that sense it can be considered a bad design, as it is hard to reuse. Conveniently, the algorithm can be optimized to use only 2 adders, no matter the window length. Listing 1.11 shows that instead of summing all the elements, the overlapping part of previous calculation can be used to significantly optimize the algorithm.

Listing 1.11: Accumulator

```
y[4] = x[4] + x[5] + x[6] + x[7] + x[8] + x[9]
y[5] =        x[5] + x[6] + x[7] + x[8] + x[9] + x[10]
y[6] =               x[6] + x[7] + x[8] + x[9] + x[10] + x[11]

# reusing overlapping parts implementation
y[5] = y[4] + x[10] - x[4]
y[6] = y[5] + x[11] - x[5]
```

Listing 1.12 gives the implementation of optimal sliding adder, it features new register `sum` that keeps track of the previous output. Note that the `shr` stayed the same, but is now rather used as a delay-chain.

Listing 1.12: Optimal sliding adder

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

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

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

Fig. 1.12 shows the synthesis result, as expected, critical path is 2 adders.

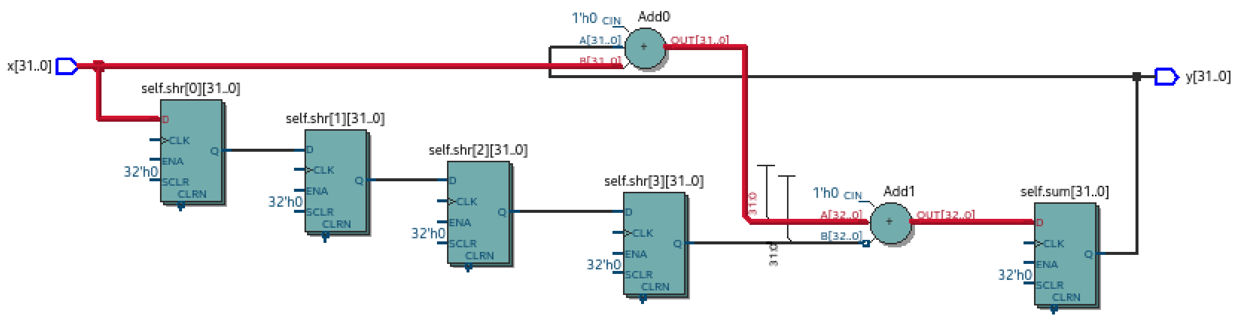


Fig. 1.12: Synthesis result of Listing 1.9, `window_len=4` (Intel Quartus RTL viewer)

Simulations results(Fig. 1.13) show that the hardware desing behaves exactly as the software model. Note that the class has `self._delay=1` to compensate for the register delay.

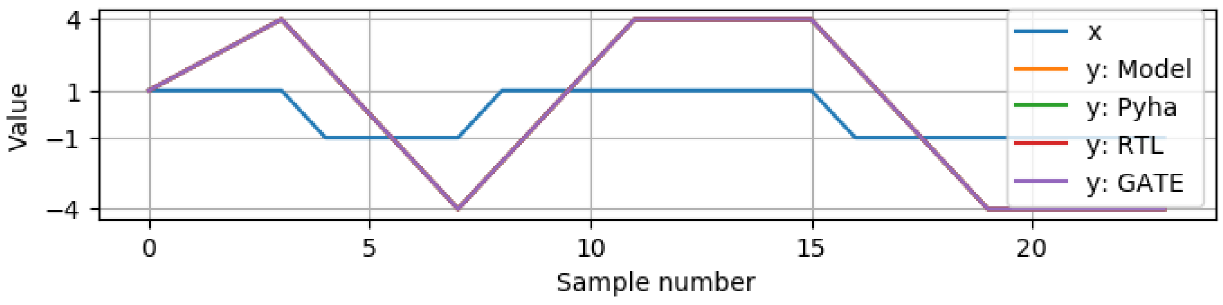


Fig. 1.13: Simulation results for `OptimalSlideAdd(window_len=4)`

1.3.3 Conclusion

In Pyha all class variables are interpreted as hardware registers. The `__init__` function may contain any Python code to evaluate reset values for registers.

Key difference between software and hardware approach is that hardware registers have **delayed assignment**, they must be assigned to `self.next`.

The delay introduced by the registers may drastically change the algorithm, thats why it is important to always have a model and unit tests, before starting hardware implementation. Model delay can be specified by `self._delay` attribute, this helps the simulation functions to compensate for the delay.

Registers are also used to shorten the critical path or logic elements, thus allowing higher clock rate. It is encouraged to register all the outputs of Pyha designs.

1.4 Fixed-point designs

Examples on the previous chapters have used only the `integer` type, in order to simplify the designs.

DSP algorithms are described using floating point numbers. As shown in previous sections, every operation in hardware takes resources and floating point calculations cost greatly. For that reason, in hardware world it is more common to use fixed-point arithmetic instead.

Fixed-point arithmetic is in nature equal to integer arithmetic and thus can use the DSP blocks that come with many FPGAs (some high-end FPGAs have also floating point DSP blocks *[14]*).

1.4.1 Basics

Pyha defines `Sfix` for FP objects, it is an signed number. It works by defining bits designated for `left` and `right` of the decimal point. For example `Sfix(0.3424, left=0, right=-17)` has 0 bits for integer part and 17 bits for fractional part. Listing 1.13 shows some examples. more information about the fixed point type is given on APPENDIX.

Todo

Add more information about fixed point stuff to the appendix

Listing 1.13: Fixed point precision

```
>>> Sfix(0.3424, left=0, right=-17)
0.34239959716796875 [0:-17]
>>> Sfix(0.3424, left=0, right=-7)
0.34375 [0:-7]
>>> Sfix(0.3424, left=0, right=-4)
0.3125 [0:-4]
```

Default FP type in Pyha is `Sfix(left=0, right=-17)`, it represents numbers between [-1;1] with resolution of 0.000007629. This format is chosen because it fits into common FPGA DPS blocks (18 bit signals *[13]*) and it can represent normalized numbers.

General recommendation is to keep all the inputs and outputs of the block in the default type.

1.4.2 Fixed-point sliding adder

Consider converting the sliding window adder to FP implementation. This requires changes only in the `__init__` function (Listing 1.14).

Listing 1.14: Fixed-point sliding adder

```
def __init__(self, window_size):
    self.shr = [Sfix()] * window_size
    self.sum = Sfix(left=0)
...
```

First line sets `self.mem` to store `Sfix()` elements. Notice that it does not define the fixed-point bounds, meaning it will store 'whatever' is assigned to it. Final bounds are determined during simulation.

The `self.sum` register uses another lazy statement of `Sfix(left=0)`, meaning that the integer bits are forced to 0 bits on every assign to this register. Fractional part is left openly determined during simulation. Rest of the code is identical to the 'integer' version.

Synthesis results are shown on Fig. 1.14. In general the RTL looks familiar to the version that used `integer` types. First noticable change is that the signals are now 18 bits wide due to the default FP type. Second big addition is the saturation logic, which prevents the wraparound behaviour by forcing the maximum or negative value when out of fixed point format. Saturation logic is by default enabled for FP types.

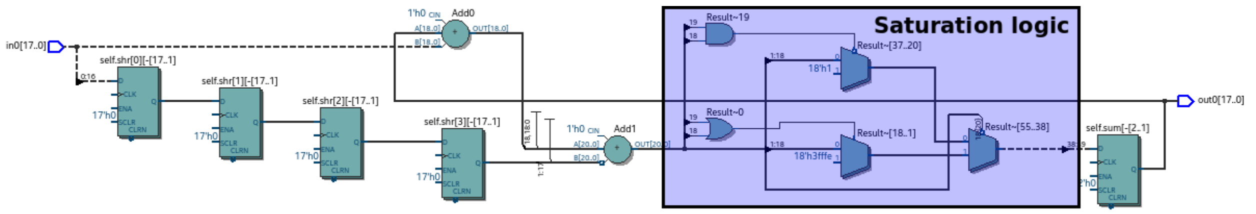


Fig. 1.14: RTL with saturation logic (Intel Quartus RTL viewer)

Fig. 1.15 plots the simulation results. Notice that the hardware simulations are bounded to [-1;1] range by the saturation logic, that is why the model simulation is different at some parts.

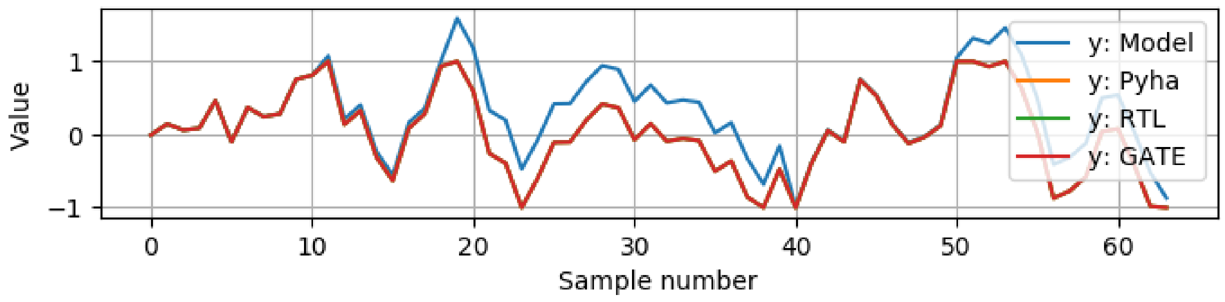


Fig. 1.15: Simulation results of FP sliding sum

Simulation functions can automatically convert 'floating-point' inputs to default FP type. In same manner, FP outputs are converted to floating point numbers. That way designer does not have to deal with FP numbers in unit-testing code. Example is given on Listing 1.15.

Listing 1.15: Test fixed-point design with floating-point numbers

```
dut = OptimalSlidingAddFix(window_len=4)
x = np.random.uniform(-0.5, 0.5, 64)
y = simulate(dut, x)
# plotting code ...
```

1.4.3 Moving average filter

The moving average (MA) is the most common filter in DSP, mainly because it is the easiest digital filter to understand and use. In spite of its simplicity, the moving average filter is optimal for a common task: reducing random noise while retaining a sharp step response. This makes it the premier filter for time domain encoded signals *[15]*.

Fig. 1.16 shows that MA is an good algorithm for noise reduction. Increasing the window length reduces more noise but also increases the complexity and delay of the system (MA is a special case of FIR filter, same delay semantics apply).

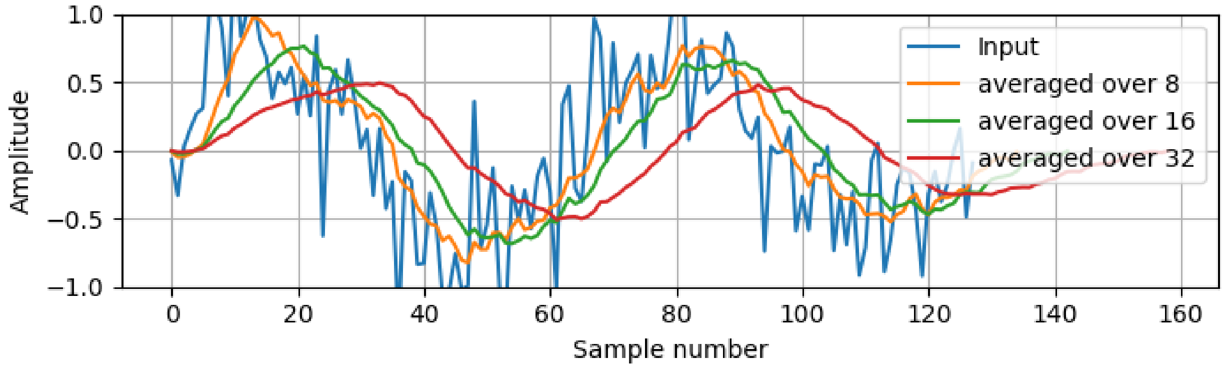


Fig. 1.16: MA algorithm in removing noise

Good noise reduction performance can be explained by the frequency response of MA (Fig. 1.17), showing that it is a low-pass filter. Passband width and stopband attenuation are controlled by the window length.

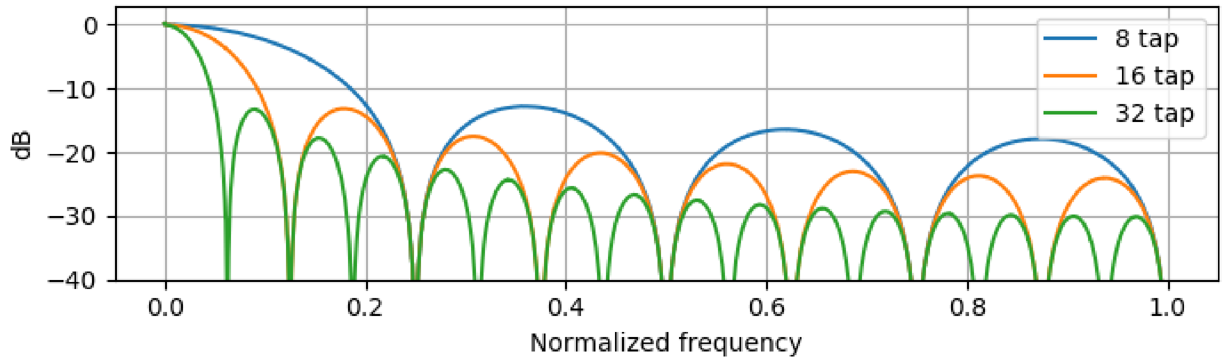


Fig. 1.17: Frequency response of MA filter

Implementation in Pyha

MA is implemented by using an sliding sum, that is divided by the sliding window length. Sliding sum part has already been implemented in previous chapter. The division can be implemented by shift right if divisor is power of two.

In addition, division can be performed on each sample instead of on the sum, that is `(a + b) / c = a/c + b/c`. Doing this guarantees that the `sum` variable is always in [-1;1] range, thus the saturation logic can be removed.

Listing 1.16: MA implementation in Pyha

```python
class MovingAverage(HW):
    def __init__(self, window_len):
```

```
3          self.window_pow = Const(int(np.log2(window_len)))
4
5          self.mem = [Sfix()] * window_len
6          self.sum = Sfix(0, 0, -17, overflow_style=fixed_wrap)
7          self._delay = 1
8
9      def main(self, x):
10         div = x >> self.window_pow
11
12         self.next.mem = [div] + self.mem[:-1]
13         self.next.sum = self.sum + div - self.mem[-1]
14         return self.sum
15     ...
```

Code on Listing 1.16 makes only few changes to the sliding sum:

- On line 3, `self.window_pow` stores the bit shift count (to support generic `window_len`)
- On line 6, type of `sum` is changed so that saturation is turned off
- On line 10, shift operator performs the division

Fig. 1.18 shows the synthesized result of this work, as expected it looks very similar to the sliding sum RTL. In general, shift operators are hard to notice on the RTL graphs because they are implemented by routing semantics.

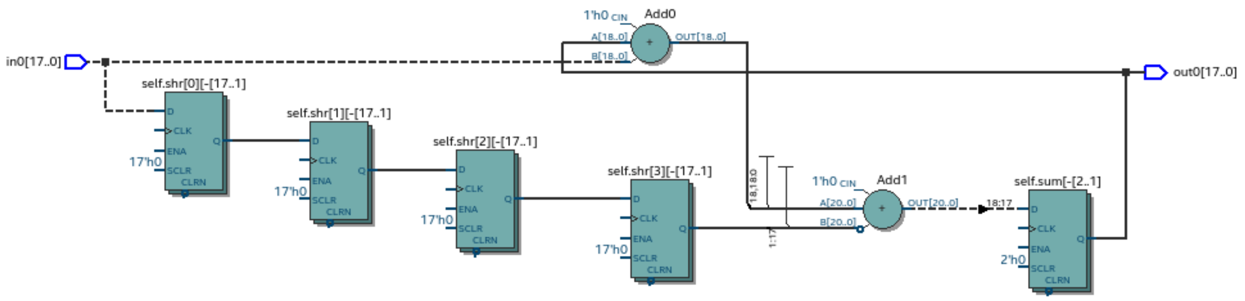


Fig. 1.18: RTL view of moving average (Intel Quartus RTL viewer)

Simulation/Testing

MA is an optimal solution for performing matched filtering of rectangular pulses *[15]*. This is important for communication systems, Fig. 1.19 shows an example of (a) digital signal is corrupted with noise. MA with window length equal to samples per symbol recovering the signal from the noise (b).

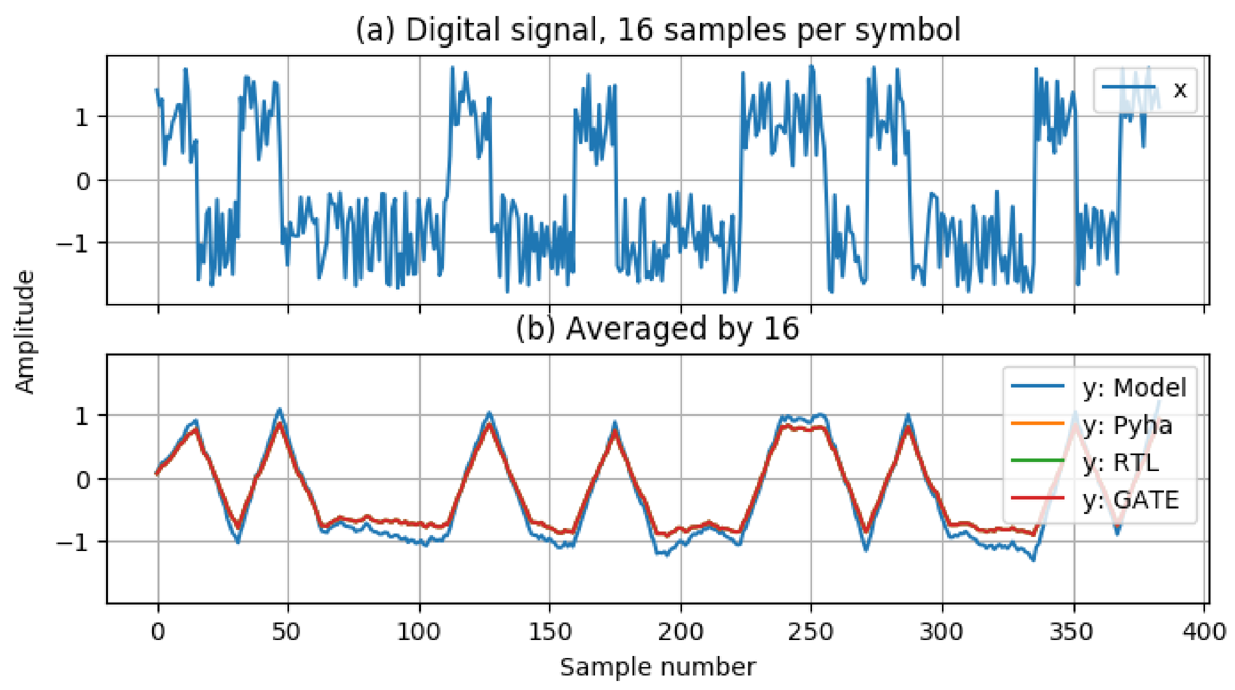


Fig. 1.19: Moving average as matched filter

The ‘model’ deviates from rest of the simulations because the input signal violates the [-1;1] bounds and hardware simulations are forced to saturate the values.

1.4.4 Conclusion

In Pyha, DSP systems can be implemented by using the fixed-point type. The combination of ‘lazy’ bounds and default Sfix type provide simplified conversion from floating point to fixed point. In that sense it could be called ‘semi-automatic conversion’.

Simulation functions can automatically perform the floating to fixed point conversion, this enables writing unit-tests using floating point numbers.

Comparing the FP implementation to the floating-point model can greatly simplify the final design process.

1.5 Abstraction and Design reuse

Pyha is based on the object-oriented design practices, this greatly simplifies the design reuse as the classes can be used to initiate objects. Another benefit is that classes can abstract away the implementation details, in that sense Pyha can become High-Level Synthesis (HLS) language.

This chapter gives an example on how to reuse the moving average filter.

1.5.1 Linear-phase DC removal Filter

Direct conversion (homodyne or zero-IF) receivers have become very popular recently especially in the realm of software defined radio. There are many benefits to direct conversion receivers, but there are also some serious drawbacks, the largest being DC offset and IQ imbalances *[16]*.

DC offset looks like a peak near the 0Hz on the frequenscy response. In time domain, it manifests as a constant component on the harmonic signal.

In *[17]*, Rick Lyons investigates the use of moving average algorithm as a DC removal circuit. This works by subtracting the MA output from the input signal. Problem of this approach is the 3 dB passband ripple. However, by connecting multiple stages of MA's in series, the ripple can be avoided (Fig. 1.20) *[17]*.

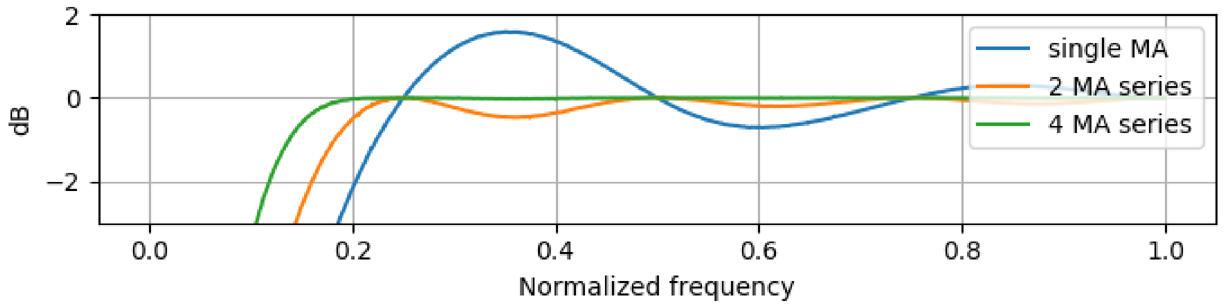


Fig. 1.20: Frequency response of DC removal filter (MA window length is 8)

Implementation

The algorithm is composed of two parts. First, four MA's are connected in series, outputting the DC component of the signal. Second the MA's output is subtracted from the input signal, thus giving the signal without DC component. Listing 1.17 shows the Pyha implementation.

Listing 1.17: DC-Removal implementation

```
class DCRemoval(HW):
    def __init__(self, window_len):
        self.mavg = [MovingAverage(window_len), MovingAverage(window_len),
                     MovingAverage(window_len), MovingAverage(window_len)]
        self.y = Sfix(0, 0, -17)

        self._delay = 1
    def main(self, x):
        # run input signal over all the MA's
        tmp = x
        for mav in self.mavg:
            tmp = mav.main(tmp)

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

This implementation is not exactly following the *[17]*. They suggest to delay-match the step 1 and 2 of the algorithm, but since we can assume the DC component to be more or less stable, it does not matter.

Fig. 1.21 shows that the synthesis generated 4 MA filters that are connected in series, output of this is subtracted from the input.

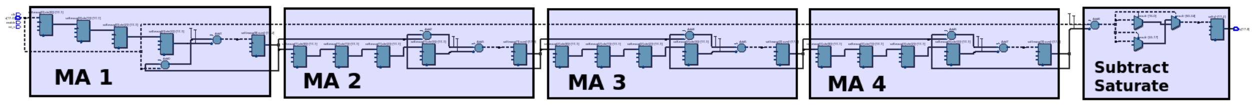


Fig. 1.21: Synthesis result of `DCRemoval(window_len=4)` (Intel Quartus RTL viewer)

In real application, one would want to use this component with larger `window_len`. Here 4 was chosen to keep the RTL simple. For example, using `window_len=64` gives much better cutoff frequency (Fig. 1.22), FIR filter with the same performance would require hundreds of taps *[17]*. Another benefit is that this filter delays the signal by only 1 sample.

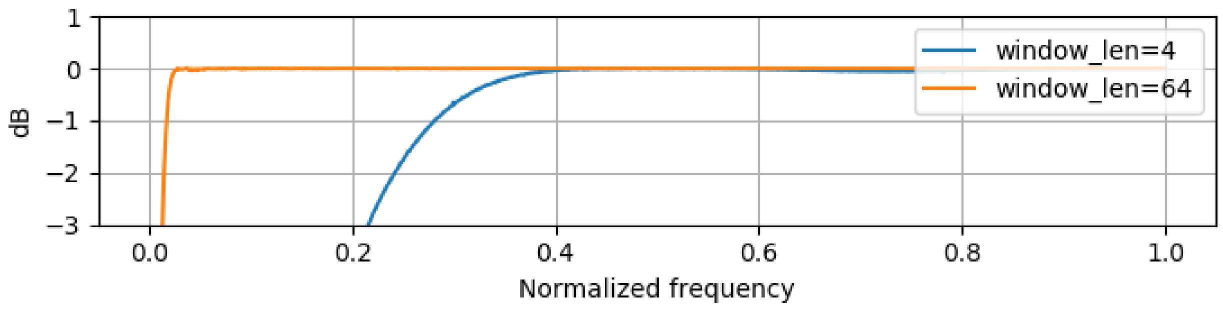


Fig. 1.22: Comparison of frequency response

This implementation is also very light on the FPGA resource usage (Listing 1.18).

Listing 1.18: Cyclone IV FPGA resource usage (`window_len = 64`)

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

Testing

Fig. 1.23 shows the situation where the input signal is corrupted with DC component (+0.25), the output of the filter starts countering the DC component until it is removed.

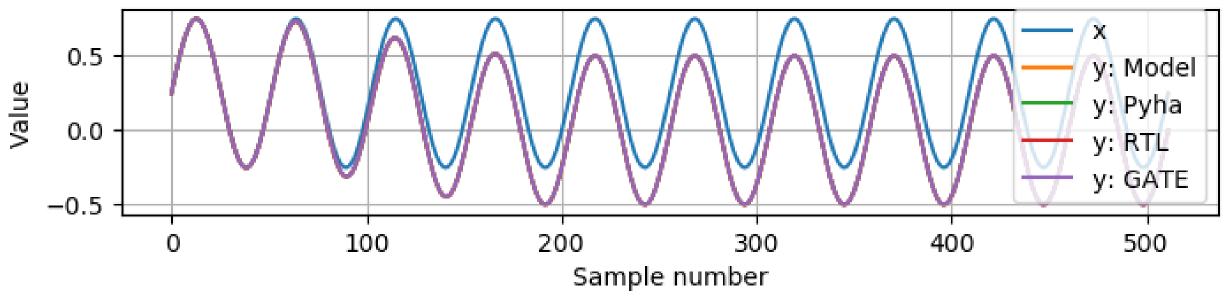


Fig. 1.23: Simulation of DC-removal filter in time domain

1.6 Conclusion

This chapter has demonstrated that in Pyha traditional software language features can be used to infer hardware components and the output of them are equal. One must still keep in mind of how the code converts to hardware, for example that the loops will be unrolled. Big difference between hardware and software is that in hardware, every arithmetical operator takes up resources.

Class variables can be used to add memory to the design. In Pyha, class variables must be assigned to `self.next` as this mimics the **delayed** nature of registers. General rule is to always register the outputs of Pyha designs.

DSP systems can be implemented by using the fixed-point type. Pyha has 'semi-automatic conversion' from floating point to fixed point numbers. Verifying against floating point model helps the iteration speed.

Reusing Pyha designs is easy thanks to the object-oriented style that also works well for design abstraction.

Pyha provides `simulate` function that can automatically run Model, Pyha, RTL and GATE level simulations. In addition, `assert_simulate` can be used for fast design of unit-tests.

These functions can automatically handle fixed point conversion, so that tests do not have to include fixed point semantics. Pyha designs are debuggable in Python domain.

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 *[1]*.

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 *[1]*.

Jiri Gaisler has proposed a 'Structured VHDL design method *[1]*' 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 *[1]*.

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 *[21]* a proposal was made to extend the VHDL language with OOP semantics (dataflow based), this effort ended with the development of OO-VHDL *[22]*, 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 *[23]* and software side of hardware design *[24]*. 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.

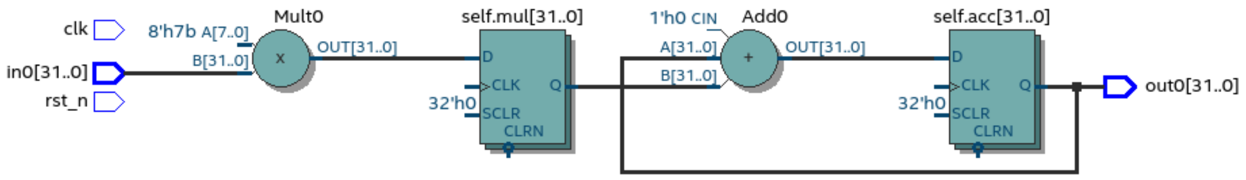


Fig. 2.1: Synthesis result of Listing 2.1 (Intel Quartus RTL viewer)

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
```

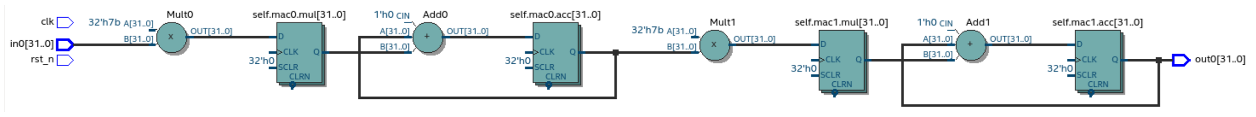


Fig. 2.2: Synthesis result of Listing 2.2 (Intel Quartus RTL viewer)

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
```

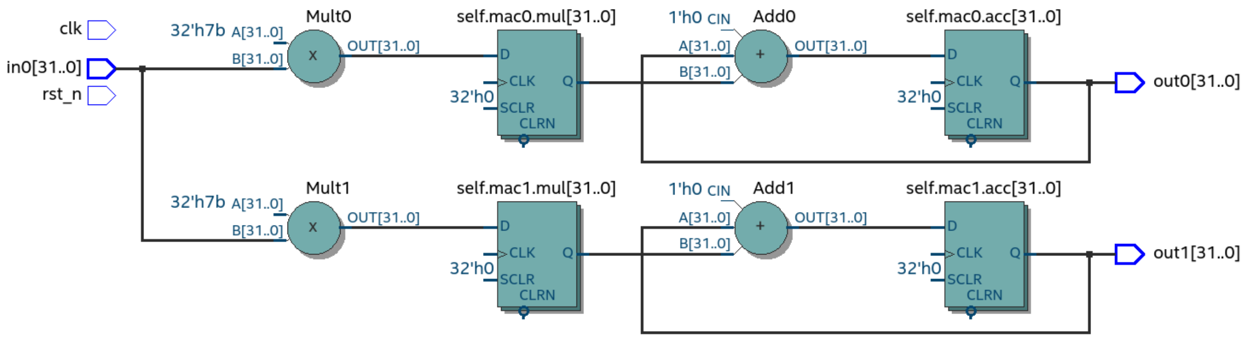


Fig. 2.3: Synthesis result of Listing 2.3 (Intel Quartus RTL viewer)

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'.

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 *[25]*. 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 *[26]*. 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 *[27]*, 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 *[28]*.

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’ *[29]*, 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

```
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)

```
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.

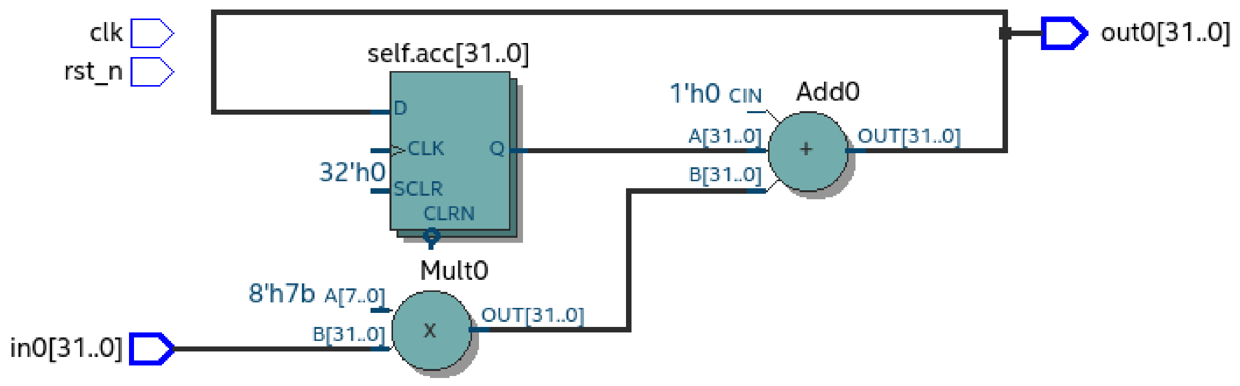


Fig. 2.4: Synthesis result of Listing 2.5 (Intel Quartus RTL viewer)

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 *[23]*.

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 *[30]*. The same semantics are also used in MyHDL *[23]*.

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

```
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.

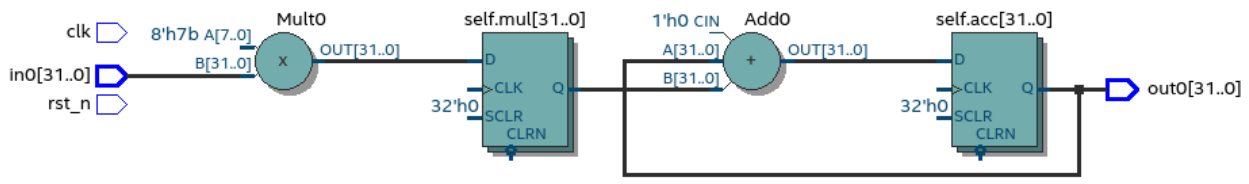


Fig. 2.5: Synthesis result of the revised code (Intel Quartus RTL viewer)

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 *[29]*. 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;
```

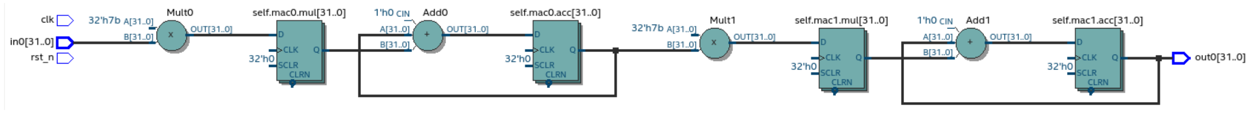


Fig. 2.6: Synthesis result of the new class (Intel Quartus RTL viewer)

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;
```

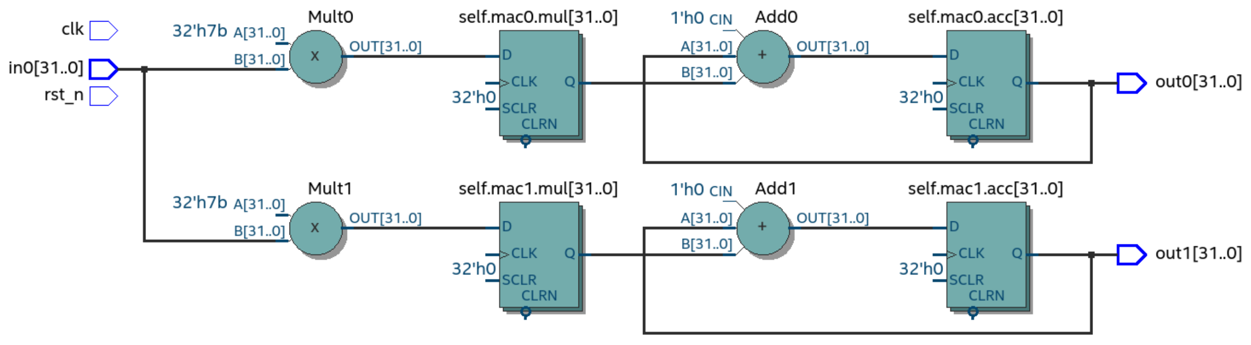


Fig. 2.7: Synthesis result of Listing 2.15 (Intel Quartus RTL viewer)

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

```
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;
```

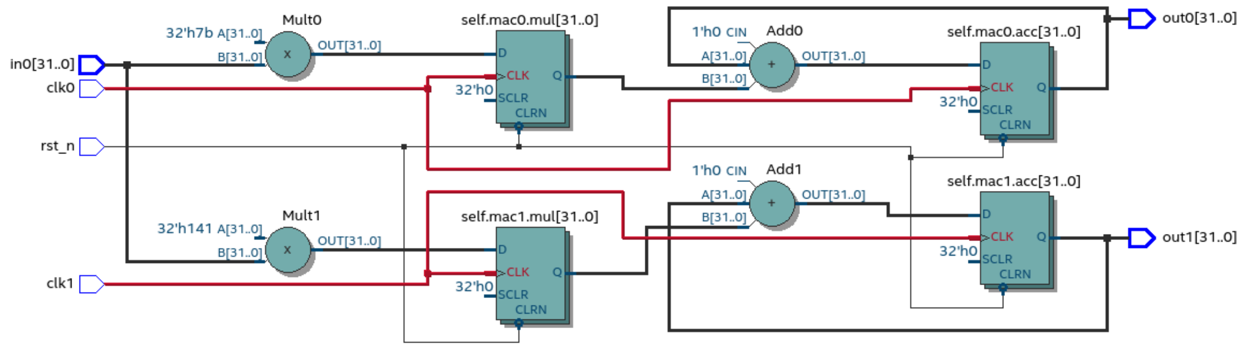


Fig. 2.8: Synthesis result with modified top-level process (Intel Quartus RTL viewer)

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

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

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

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

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

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

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

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

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

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

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

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

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

[14] Amulya Vishwanath. Enabling high-performance floating-point designs. URL: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01267-fpgas-enable-high-performance-floating-point.pdf.

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

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

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

[18] 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-/.

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

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

[21] 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.

[22] 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.

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

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

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

[26] 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/.

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

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

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

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