Pyha Documentation

Release 0.0.0

Gaspar Karm

Mar 28, 2017

CONTENTS:

CHAPTER

ONE

INTRODUCTION

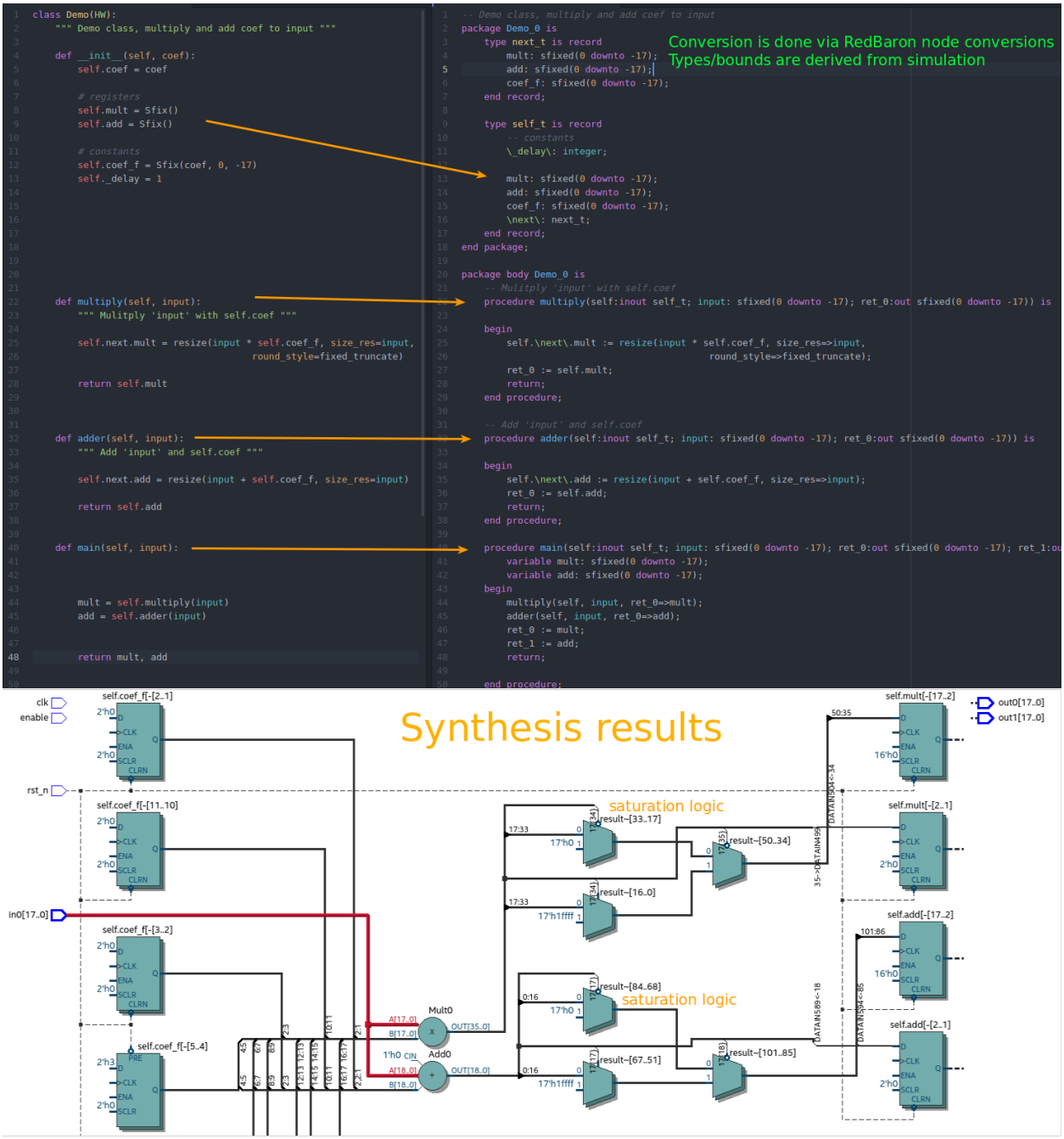
Essentially this is a Python to VHDL converter, with a specific focus on implementing DSP systems.

Main features:

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

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

Working principle



As shown on above image, Python sources are turned into synthesizable VHDL code. In `__init__`, any valid Python code can be used, all the variables are collected as registers. Objects of other classes (derived from HW) can be used as registers, even lists of objects is possible.

In addition, there are tools to help verification by automating RTL and GATE simulations.

Limitations/future work

Currently designs are limited to one clock signal, decimators are possible by using Streaming interface. Future plans is to add support for multirate signal processing, this would involve automatic PLL configuration. I am thinking about integration with Qsys to handle all the nasty clocking stuff.

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

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

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

Credits

Inspiration:

- A Structured VHDL Design Method: Shows how to do structured VHDL by suggesting only two processed design with the use of functions and structures. Pyha takes this idea to extreme, by using only 1 procedure and 1 entity per whole design.
- MyHDL: It is great! I started from scratch because i wanted to try higher level approach for the conversion.

Essential components:

- RedBaron: Enables conversion from Python to VHDL.
- GHDL: Open VHDL simulator.
- Cocotb: Python to simulator communications.
- PyTest: Unit testing.

CHAPTER

TWO

INSTALLATION

Note: Pyha works only on Python 3.6 and currently is developed/tested on Ubuntu 14/16.

To install pyha:

```
pip install pyha
```

RTL-level simulations

GHDL and Cocotb are required to run RTL simulations.

Install GHDL:

```
wget https://github.com/tgingold/ghdl/releases/download/2016-09-14/ghdl-0.34dev-mcode-
↪2016-09-14.tgz -O /tmp/ghdl.tar.gz
mkdir ghdl
tar -C ghdl -xvf /tmp/ghdl.tar.gz

# add GHDL to path
echo export PATH=$PWD/ghdl/bin/:$PATH >> ~/.bashrc
source ~/.bashrc
```

Cocotb must be installed from fork (it includes some Python3.6 overwrites). Install Cocotb:

```
sudo apt-get install git make gcc g++ swig
git clone https://github.com/petspats/cocotb

# set COCOTB path
echo export COCOTB=$PWD/cocotb >> ~/.bashrc
source ~/.bashrc
```

GATE-level simulations

Install Intel Quartus ,make sure that you enable Cyclone IV support.

After installing, you can build GHDL support libraries:

```
python scripts/compile_quartus_lib.py
```

It is normal that is 'fails':

```
--------------------------------------------------------------------------------
Compiling Altera Quartus libraries [FAILED]
```

At this point you can **optionally** run tests, be warned that it takes up to 30 minutes.

```
pytest tests/
```

CHAPTER

THREE

FIXED-POINT

Pyha maps fixed-point operations almost directly to VHDL fixed point library

class `pyha.common.sfix.`**`Sfix`**(*val=0.0*, *left=0*, *right=0*, *init_only=False*, *overflow_style='fixed_saturate'*, *round_style='fixed_round'*)

Signed fixed point type, like to_sfixed() in VHDL. Basic arithmetic operations are defined for this class.

More info: https://www.dsprelated.com/showarticle/139.php

Parameters

- **`val`** – initial value
- **`left`** – bits for integer part.
- **`right`** – bits for fractional part. This is negative number.
- **`init_only`** – internal use only
- **`overflow_style`** – fixed_saturate(default) or fixed_wrap
- **`round_style`** – fixed_round(default) or fixed_truncate

```
>>> Sfix(0.123, left=0, right=-17)
0.1230010986328125 [0:-17]
>>> Sfix(0.123, left=0, right=-7)
0.125 [0:-7]
```

```
>>> Sfix(2.5, left=0, right=-17)
WARNING:pyha.common.sfix:Saturation 2.5 -> 0.9999923706054688
0.9999923706054688 [0:-17]
>>> Sfix(2.5, left=1, right=-17)
WARNING:pyha.common.sfix:Saturation 2.5 -> 1.9999923706054688
1.9999923706054688 [1:-17]
>>> Sfix(2.5, left=2, right=-17)
2.5 [2:-17]
```

static `set_float_mode`(*x*)

Can be used to turn off all quantization effects, useful for debugging.

Parameters **`x`** – True/False

Complex numbers

Puha supports complex numbers for interfacing means, arithmetic operations are not defined. Use `.real` and `.imag` to do maths.

class pyha.common.sfix.**ComplexSfix**(*val=0j*, *left=0*, *right=0*, *overflow_style='fixed_saturate'*)

Use real and imag members to access underlying Sfix elements.

Parameters

- **val** –
- **left** – left bound for both components
- **right** – right bound for both components
- **overflow_style** – fixed_saturate(default) or fixed_wrap

```
>>> a = ComplexSfix(0.45 + 0.88j, left=0, right=-17)
>>> a
0.45+0.88j [0:-17]
>>> a.real
0.4499969482421875 [0:-17]
>>> a.imag
0.8799972534179688 [0:-17]
```

Another way to construct it:

```
>>> a = Sfix(-0.5, 0, -17)
>>> b = Sfix(0.5, 0, -17)
>>> ComplexSfix(a, b)
-0.50+0.50j [0:-17]
```

Utility functions

Most of the arithmetic functions are defined for Sfix class. Sizing rules known from VHDL fixed point library apply.

pyha.common.sfix.**resize**(*fix*, *left_index=0*, *right_index=0*, *size_res=None*, *overflow_style='fixed_saturate'*, *round_style='fixed_round'*)

Resize fixed point number.

Parameters

- **fix** – Sfix object to resize
- **left_index** – new left bound
- **right_index** – new right bound
- **size_res** – provide another Sfix object as size reference
- **overflow_style** – fixed_saturate(default) or fixed_wrap
- **round_style** – fixed_round(default) or fixed_truncate

Returns New resized Sfix object

```
>>> a = Sfix(0.89, left=0, right=-17)
>>> a
0.8899993896484375 [0:-17]
>>> b = resize(a, 0, -6)
>>> b
0.890625 [0:-6]
```

```
>>> c = resize(a, size_res=b)
>>> c
0.890625 [0:-6]
```

pyha.common.sfix.**left_index**(*x: pyha.common.sfix.Sfix*)

Use this in convertible code

Returns left bound

```
>>> a = Sfix(-0.5, 1, -7)
>>> left_index(a)
1
```

pyha.common.sfix.**right_index**(*x: pyha.common.sfix.Sfix*)

Use this in convertible code

Returns right bound

```
>>> a = Sfix(-0.5, 1, -7)
>>> right_index(a)
-7
```

CHAPTER

FOUR

TUTORIAL:BASIC USAGE

In this tutorial we will write a simple fixed point multiplier. It serves as a basic example of turning DSP models into HDL in sensible and testable way.

Source code

VHDL conversion

Model and unit-tests

Lets follow the model and test-driven development, it goes like this:

- Write simplest possible model of what you want to do
- Experiment with it, dont throw away the experiments, rather turn them into unit-tests
- Profit (use the same tests to develop and verify HDL code)

Start by defining an model:

```
from pyha.common.hwsim import HW
import numpy as np


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

    def model_main(self, input_list):
        # note that model works on lists
        return np.array(input_list) * self.coef
```

Note: `model_main` function is reserved for defining the model. It is not convertable to VHDL, we use it to verify RTL against this.

Note: Pyha operates on classes, they must be derived from pyha.HW.

Next step is to write all the tests we can imagine, later we can use the same tests on RTL code.

One possible test, that pushes some data into the model and compares the output with the expected output.

```
def test_basic():
    from pyha.simulation.simulation_interface import SIM_MODEL, assert_sim_match, SIM_
↪HW_MODEL, SIM_RTL, SIM_GATE
    dut = BigFir(coef=0.5)
    inputs = [0.1, 0.2, 0.3, 0.2, 0.1]
    expect = [0.05, 0.1, 0.15, 0.1, 0.05]

    assert_sim_match(dut, None, expect, inputs, simulations=[SIM_MODEL])
```

pyha.simulation.simulation_interface.**assert_sim_match**(*model*, *types*, *expected*, **x*, *simulations=None*, *rtol=1e-05*, *atol=1e-09*, *dir_path=None*, *skip_first=0*)

Run bunch of simulations and assert that they match outputs.

Parameters

- **model** – Instance of your class
- **types** – If `main` is defined, provide input types for each argument, all arguments will be automatically casted to these types.
- **expected** – Expected output of the simulation. If None, assert all simulations match eachother.
- **x** – Inputs, if you have multiple inputs, use *x for unpacking.
- **simulations** – Simulations to run and assert:

•SIM_MODEL: runs model ('model_main')

•SIM_HW_MODEL: runs HW model ('main')

•SIM_RTL: converts to VHDL and runs RTL simulation via GHDL and Cocotb

•SIM_GATE: runs sources trough Quartus and simulates the generated netlist

---

Note: If None(default), runs all simulations. SIM_HW_MODEL must be run if SIM_RTL or SIM_GATE are going to run.

Parameters

- **rtol** – Relative tolerance for assertion. Look np.testing.assert_allclose.
- **atol** – Absolute tolerance for assertion. Look np.testing.assert_allclose.
- **dir_path** – Where are conversion outputs written, if empty uses temporary directory.
- **skip_first** – Skip first 'n' samples for assertion.

Note: If you use PyCharm you can run unit-tests by right clicking on the function name and selecting 'Run py.test..' You may need to set `File-Settings-Tools-Python Integrated Tools-Default Test Runner = py.test`. You can also run PyTest from console `$ pytest`

Hardware model

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

```
1  from pyha.common.const import Const
2  from pyha.common.sfix import Sfix, resize, fixed_truncate
3  from pyha.common.hwsim import HW
4  import numpy as np
5
6
7  class BigFir(HW):
8      def __init__(self, coef):
9          self.coef = coef
10
11         # define output registers
12         # bounds will be determined during simulation
13         self.out_resized = Sfix()
14
15         # constants
16         self.coef_f = Sfix(coef, 0, -17)
17
18         # uncomment this and quartus will optimize away multiplication (assuming
   ↪coef=0.5)
19         # self.coef_f = Const(Sfix(coef, 0, -17))
20
21     def main(self, input):
22         # this will also infer saturation logic
23         # for registers you always assign to self.next
24         self.next.out_resized = resize(input * self.coef_f, size_res=input,
25                                        round_style=fixed_truncate)
26
27         return self.out_resized
28
29     def model_main(self, input_list):
30         # note that model works on lists
31         return np.array(input_list) * self.coef
```

In Line 13, we defined a register named `out_resized`. It is using lazy-Sfix notation, meaning that the actual bounds are derived from the data you feed into the model.

Note: All the class variables are interpreted as registers, unconvertable types like float or Numpy arrays will be ignord for conversion. All the assignments to registers go trough `self.next`

Line 16 turns the floating point coef into fixed-point.

Note: `main` function is reserved for defining the HDL model, this is convertable to VHDL. This is the main entry to the model, you can call other functions if needed.

On line 23 resized result of multiplication is assigned to register. This also infers saturation logic.

Results are returned on line 25, multiple values can be returned in Pyha.

Testing

Only minor modifications are required to adapt the test function:

```
1  def test_basic():
2      from pyha.simulation.simulation_interface import SIM_MODEL, assert_sim_match, SIM_
   ↪HW_MODEL, SIM_RTL, SIM_GATE
3      dut = BigFir(coef=0.5)
4      inputs = [0.1, 0.2, 0.3, 0.2, 0.1]
5      expect = [0.05, 0.1, 0.15, 0.1, 0.05]
6
7      assert_sim_match(dut,
8                       [Sfix(left=0, right=-17)],
9                       expect, inputs,
10                      simulations=[SIM_MODEL, SIM_HW_MODEL])
```

On line 8 we added the input signature of our 'main' function and on line 10 we added a HW simulation instruction.

Note: `SIM_HW_MODEL` is Python based simulation, you can use debugger to see how your 'main' function is called. Debugger is quite an useful tool in Pyha designs since everything is fully sequentially executed.

Note: You can write models in such way that input signature determines the output types. Your VHDL conversion will depend on this.

Upon running the test:

```
INFO:Running MODEL simulation!
INFO:Running HW_MODEL simulation!
ERROR:#################################################################
ERROR:#################################################################
ERROR:               "HW_MODEL" failed
ERROR:#################################################################
ERROR:#################################################################

... stack trace ...
```

```
AssertionError:
Not equal to tolerance rtol=1e-05, atol=1e-09
E
(mismatch 100.0%)
x: array([ 0.05,  0.1 ,         0.15,       0.05,       0.1])
y: array([ 0.  ,  0.050003,     0.099998,   0.150002,   0.099998])
```

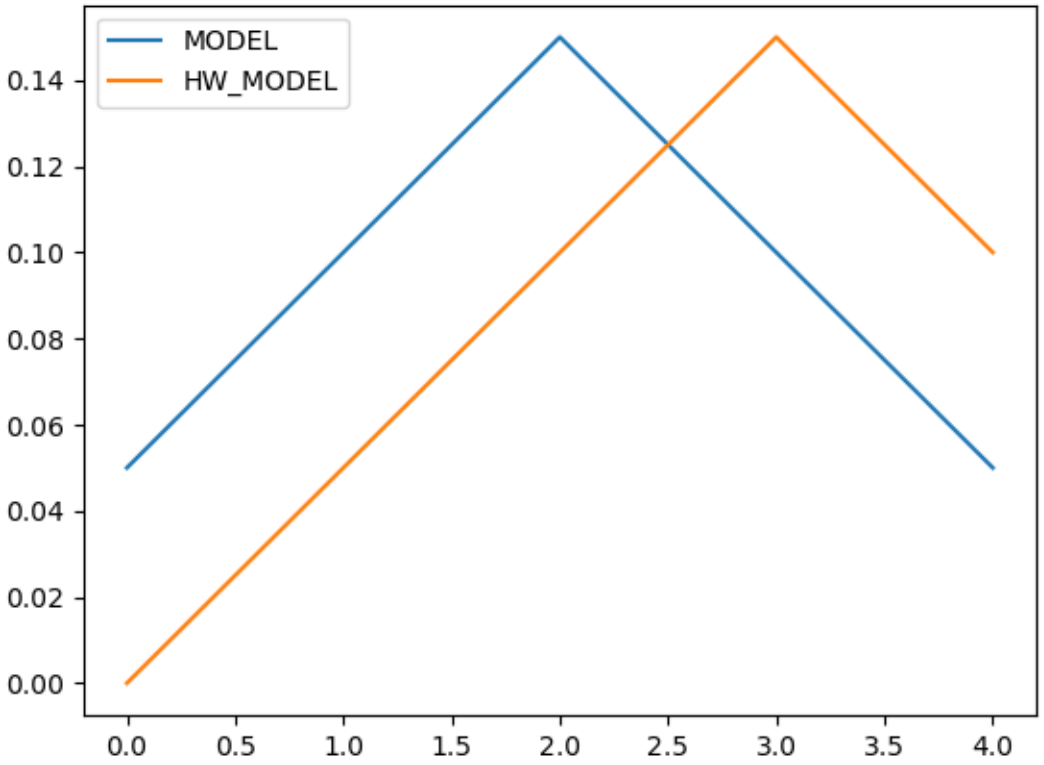
Hardware simulation failed, looking closely reveals the expected and actual outputs are just delayed by 1.

Alternatively you can use a debug function:

pyha.simulation.simulation_interface.**plot_assert_sim_match**(*model*, *types*, *expected*, **x*, *simulations=None*, *rtol=1e-05*, *atol=1e-09*, *dir_path=None*, *skip_first=0*)

Same arguments as `assert_sim_match`. Instead of asserting it plots all the simulations.

It would output:



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

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

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

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

```
AssertionError:
Not equal to tolerance rtol=1e-05, atol=1e-09
(mismatch 80.0%)
x: array([ 0.05,        0.1 ,       0.15,       0.05,       0.1 ])
y: array([ 0.050003,    0.099998,   0.150002,   0.050003,   0.099998])
```

Note: `rtol=1e-5` requires that ~5 digits after decimal point must match. `rtol=1e-4` would require 4 digits to match.

Now values are aligned, but the tolerances are too strict, we are using fixed-point after all. One way to solve this would be to add more bits to fixed-point type, for example `Sfix(left=0, right=-19)`. Better way is to set `rtol = 1e-4`. We want to keep 18 bit fixed-point numbers because Intel Cyclone FPGAs DSP blocks are of this size.

In general i am okay when simulations pass `rtol=1e-3`. You may need to adjust `atol`, when failing numbers are close to 0.

Here is the final code that passes assertions:

```
from pyha.common.const import Const
from pyha.common.sfix import Sfix, resize, fixed_truncate
from pyha.common.hwsim import HW
import numpy as np

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

        # define output registers
        # bounds will be determined during simulation
        self.out_resized = Sfix()

        # constants
        self.coef_f = Sfix(coef, 0, -17)

        # uncomment this and quartus will optimize away multiplication (assuming coef=0.5)
        # self.coef_f = Const(Sfix(coef, 0, -17))

        self._delay = 1

    def main(self, input):
        # this will also infer saturation logic
        # for registers you always assign to self.next
        self.next.out_resized = resize(input * self.coef_f, size_res=input,
                                       round_style=fixed_truncate)

        return self.out_resized

    def model_main(self, input_list):
        # note that model works on lists
        return np.array(input_list) * self.coef


def test_basic():
    from pyha.simulation.simulation_interface import SIM_MODEL, assert_sim_match, SIM_HW_MODEL, SIM_RTL, SIM_GATE
    dut = BigFir(coef=0.5)
    inputs = [0.1, 0.2, 0.3, 0.2, 0.1]
    expect = [0.05, 0.1, 0.15, 0.1, 0.05]

    assert_sim_match(dut,
                     [Sfix(left=0, right=-17)],
                     expect, inputs,
                     simulations=[SIM_MODEL, SIM_HW_MODEL],
                     rtol=1e-4)
```

RTL simulations

Add `SIM_RTL` to the simulations list.

Note: `SIM_RTL` converts sources to VHDL and runs RTL simulation by using GHDL simulator.

In case you want to view the converted VHDL files, you can use `dir_path` option.

Example:

```
assert_sim_match(dut,
                 [Sfix(left=0, right=-17)],
                 expect, inputs,
                 simulations=[SIM_MODEL, SIM_HW_MODEL, SIM_RTL],
                 dir_path='~/vhdl_conversion')
```

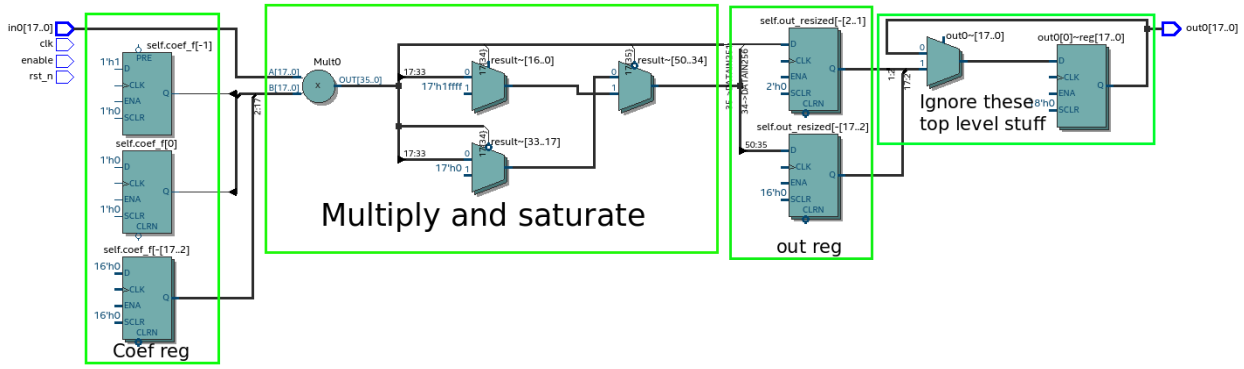
GATE simulation and Quartus

Add `SIM_GATE` to the simulations list.

Note: `SIM_GATE` runs VHDL sources trough Quartus and uses the generated generated netlist for simulation. Use to gain ~full confidence in your design. It is slow!

Running the GATE simulation, will produce 'quartus' directory in `dir_path`. One useful tool in Quartus software is RTL viewer, it can be opened from Tools-Netlist viewers-RTL viewer.

RTL of this tutorial:



Note: Design will be optimized if you mark `self.coef` as Const, Quartus will use shift instead of multiply.

CHAPTER

FIVE

EXAMPLES

Pyhacores is a repository collecting cores implemented in Pyha, for example it includes CORDIC, FSK modulator and FSK demodulator cores.

Submodules, lists and submodule lists. All sequential

Shows how multiple classes and instances can be used in structured design. Everything is sequentially executed, debugger can be used to step around in code. Itself it does nothing useful.

Source code

VHDL conversion

INDEX

P

S