Pyha

Release 0.0.0

Gaspar Karm

Mar 29, 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 Fig. 1.1, 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.

Listing 1.1: this.py

```
print('Explicit is better than implicit.')
```

See on Listing 1.1 shows print statement

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.

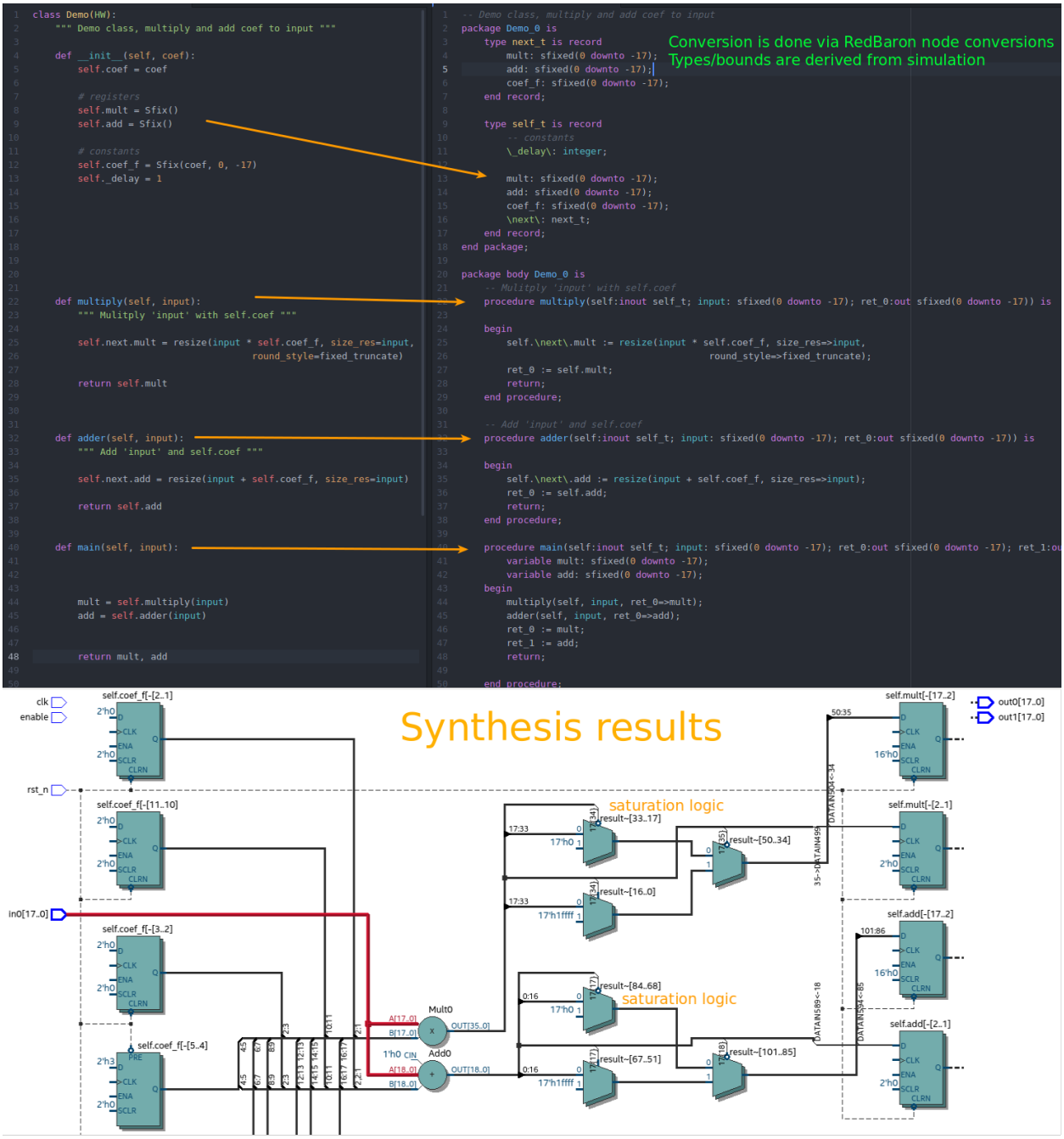

Fig. 1.1: Caption

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.

Objective/goal

Provide simpler way of turning DSP blocks to FPGA. Reduce the gap between regular programming and hardware design. Turn GNURadio flowgraphs to FPGA? Model based verification! Why do it? opensource

Scope

Focus on LimeSDR board and GnuRadio Pothos, frameworks.

Structure

First chapter of this thesis gives an short background about

CHAPTER

TWO

BACKGROUND

Give a short overview of whats up.

Python

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

HDL related tools in Python

MyHDL

Migen

CocoTb

CHAPTER

THREE

PYHA

This paragraph gives an basic overview of the developed tool.

Basics

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

Listing 3.1: Basic Pyha unit

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

    def main(self, input):
        pass

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

Listing 3.1 shows the besic design unit of the developend tool, it is a standard Python class, that is derived from a baseclass *HW, purpos of this baseclass is to do some metaclass stuff and register this class as Pyha module.

Metaclass actions:

Combinatory logic

Todo

Ref comb logic.

Listing 3.2: Basic combinatory circuit in Pyha

```
class Comb(HW):
    def main(self, a, b):
        xor_out = a xor b
        return xor_out
```

Listing 3.2 shows the design of a combinatory logic. In this case it is a simple xor operation between two input operands. It is a standard Python class, that is derived from a baseclass *HW, purpose of the baseclass is to do some metaclass stuff and register this class as Pyha module.

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

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

Todo

how this turns to VHDL and RTL picture?

Sequential logic

Todo

Ref comb logic.

Listing 3.3: Basic sequential circuit in Pyha

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

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

Listing 3.3 shows the design of a registered adder.

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

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

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

In general this system is similiar to VHDL signals:

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

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

Todo

how this turns to VHDL and RTL picture?

Types

This chapter gives overview of types supported by Pyha.

Integers

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

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

Booleans

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

Floats

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

Fixed-point type

Fixed point numbers can be to effectively turn floating point models into FPGA.

Todo

ref http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.129.5579&rep=rep1&type=pdf https://www.dsprelated.com/showarticle/139.php

---

Fixed point numbers are defined to have bits for integer size and fractional size. Integer bits determine the maximum size of the number. Fractional bits determine the minimum resolution.

Main type of Pyha is Sfix, that is an signed fixed point number.

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

Overflows and Saturation

Practical fixed-point variables can store only a part of what floating point value could. Converting a design from floatin to fixed point opens up a possiblity of overflows. That is, when the value grows bigger or smaller than the format can reprsent. This condition is known as overflow.

By default Pyha uses fixed-point numbers that have saturaton enabled, meaning that if value goes over maximum possible value, it is instead kept at the maximum value. Some examples:

```
>>> 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]
```

On the other hand, sometimes overflow can be a feature. For example, when designing free running counters. For this usages, saturation can be disabled.

```
>>> Sfix(0.9, left=0, right=-17, overflow_style=fixed_wrap)
0.9000015258789062 [0:-17]
```

```
>>> Sfix(0.9 + 0.1, left=0, right=-17, overflow_style=fixed_wrap)
-1.0 [0:-17]
```

Rounding

Conversion to VHDL

VHDL comes with a strong support for fixed-point types by providing and fixed point package in the standard library. More information is about this package is given in *[1]*.

In general Sfix type is built in such a way that all the functions map to the VHDL library, so no conversion is neccesary.

Another option would have been to implement fixed point compiler on my own, it would provide more flexibility but it would take many time + it has t be kept in mind that the VHDL library is already production-tested. Ths mapping to VHDL library seemed like the best option.

It limits the conversion to VHDL only, for example Verilog has no fixed point package in standard library.

Complex fixed-point

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

```
>>> 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
```

Lists

Metaclass

Conversion to VHDL

Simulation and verification

Python simulation

RTL simulation

Testing

CHAPTER

FOUR

DESIGN EXAMPLES

This chapter provides some example designs implemented in Pyha.

First example developes and moving-average filter.

First three examples will interatively implement DC-removal system. First design implements an simple fixed-point accumulator. Second one builds upon this and implements moving average filter. Lastly multiple moving average filters are chained to form a DC removal circuit.

Second example is an FIR filter, with reloadable switchable taps ?

Third design example shows how to chain togather already exsisting Pyha blocks to implement greater systems. In this case it is FSK receiver. This examples does not go into details.

Moving Average

Use accumulator and shift register to develope Moving Average algorithm

Linear phase DC Removal

Todo

What is DC and why to remove it?

FIR filter

Maybe skip this one?

FSK receiver

Glue blocks togather...needs explanation...

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

BIBLIOGRAPHY

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

INDEX

P