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.

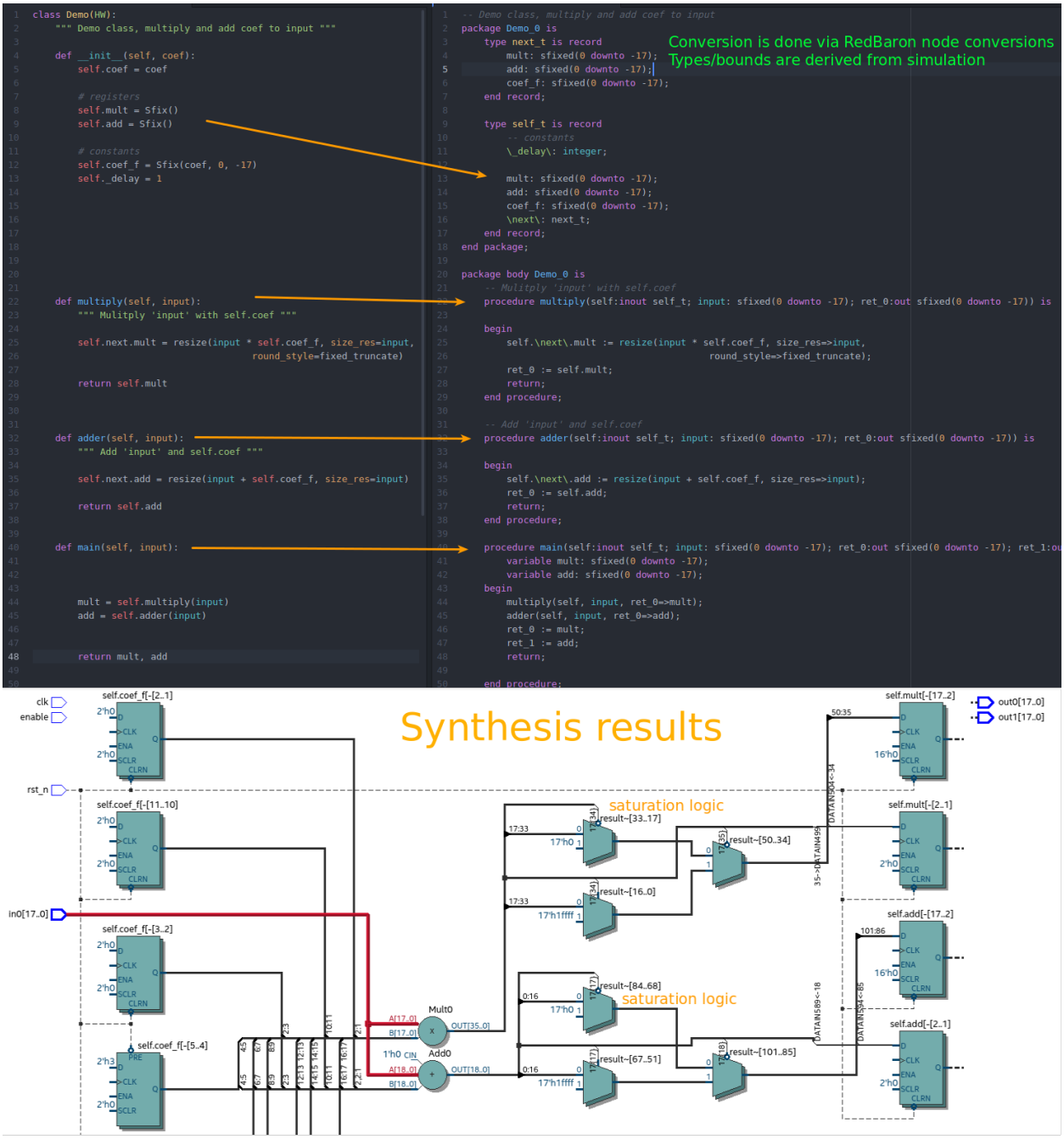

Fig. 1.1: Caption

1.2. Limitations/future work

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

How far can we go with the oneprocess design? Everyone else uses VHDL as a very low level interface.

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.

Free Dev tools. .. http://www.scipy-lectures.org/intro/intro.html#why-python

> %https://github.com/jrjohansson/scientific-python-lectures/blob/master/Lecture-0-Scientific-Computing-with-Python.ipynb

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

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

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

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

Pyha support rounding on arithmetic, basically it should be turned off as it costs alot.

Fixed-point arithmetic and sizing rules

Arithmetic operations can be run on fixed point variables as usual. Division is not defined as it is almost always unneccesary in hardware.

Library comes with sizing rules in order to guarantee that fixed point operations never overflow.

For example consider an fixed point number with format that can represent numbers between [-1, 1):

```
>>> Sfix(0.9, 0, -17)
0.9000015258789062 [0:-17]
```

Now adding two such numbers:

```
>>> Sfix(0.9, 0, -17) + Sfix(0.9, 0, -17)
1.8000030517578125 [1:-17]
```

While this operation should overflow, it did not. Because fixed point library always resizes the output for the worst case. In case of addition it always adds one integer bit to accumulate possible overflows.

But note that this system is not very smart, if we would add up such numbers 100 times, it would add 100 bits to the integer portion of the number.

The philosophy of fixed point library is to guarantee no precision loss happens during arithmetic operations, in order to do this it has to extend the output format. It is designers job to resize numbers back into optimal format after operations.

Resizing

Fixed point number can be forced to whatever size by using the resize functionality.

```
>>> 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 support automatic resizing for registers. All assignments to registers will be automatically resized to the original type of the definition.

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

Objective of this tool was to simplify model based design and verification of DSP to FPGA models. One frequent problem with DSP models was that they commonly want to use complex numbers. In order to unify the interface of the model and hardware model, Pyha supports complex numbers for interfacing means, arithmetic operations are not defined. That means complex values can be passed arond and registered but arithmetics must be done on `.real` and `.imag` elements, that are just Sfix objects.

```
>>> 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]
```

User defined types / Submodules

For design reuse it is needed to reuse previously generated designs. Traditional HDLs use entity declarations for this purpose. One of the key assumption of these entities is that they all run in parallel. This has some advantages and disadvantages. Good thing is that this is the most flexible solution, that is it supports as many clocks and clock domains as neccessary. Disadvantage is that in the end much of the VHDL programing comes down to wiring togather different entities, and this can be worksome and bugful process.

Another downside is that all of these entities must be simulated as a separate process, this has a cost on simulation speed and more severily it makes debugging hard..think about debugging multi-threaded programs.

In contrast to traditional HDLs, Pyha has taken an approach where design reuse is archived trough regular objects. This has numerous advantages:

- Defining a module is as easy as making an class object
- Using submodule is as easy as in traditional programming..just call the functions
- Execution in same domain, one process design

Result of this design decision is that using submodules is basically the same as in normal programming. This decision comes with a severe penalty aswell, namely all the submodules then must work with the same clock signal. This essentially limits Pyha designs down to using only one clock. This is a serious constrain for real life systems, but for now it can be lived with.

It is possible to get around this by using clock domain crossing interfacec between two Pyha modules.

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

Lists

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

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

For example writing `l = [1, 2]` in Python would trigger the code shown in Listing 3.4, where line 1 is a new array type definitiaon and a second line defines a variable `a` of this type. Note that the elements type is deduced from the type of first element in Python array the size of defined array is as `len(l)-1`.

Listing 3.4: VHDL conversion for integer array

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

CHAPTER

FOUR

CONVERSION

This chapter examines the feasability and means of converting Python code to VHDL.

What about verilog?

Python vs VHDL

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

Comparison of syntax

Problem of types

Biggest difference and problem between Python and VHDL is the type system. While in VHDL everything must be typed, Python is fully dynamically typed language, meaning that types only come into play when the code is executing.

In general there are some different approaches to solve this problem:

- Determining types from Python source code

How to solve dynamic typed? stuff

Language differences...

Extensions..wehn you can do more in python domain.

Feasability of converting Python to VHDL

Simulation and verification

Requrements...want RTL sim, GATE sim, in loop etc

Essentially this comes downt to being and VHDL simulator inside VHDL simulator. it may sound stupid, but it works for simulations and synthesys, so i guess it is not stupid.

Python simulation

RTL simulation

Testing

CHAPTER

FIVE

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.