blocks Documentation

Release 0.1.0

Christopher L. Felton

January 05, 2013

CONTENTS

Noble, colorless, oderless, and tasteless.

The name for this project was inspired by mercurial (i.e. laziness and lack of creativity by me). An element from the periodic table was chosen for a name: *krypton*. Th actual Python package name is **kr**.

The project contains a verification framework, mainly for signal processing (dsp) FPGA cores. This package uses MyHDL to implement the cores. Initially, the project is reimplementing the GnuRadio USRP HDL completely in Python/MyHDL. But the project is going slightly further and will make each of the DSP modules as generic and parameterizable as possible. It is the goal that the project will grow and support many different FPGA DSP systems.

CHAPTER

ONE

KRYPTON (KR)

1.1 What is krypton

Noble, colorless, oderless, and tasteless.

The name for this project was inspired by mercurial (i.e. laziness and lack of creativity by me). An element from the periodic table was chosen for a name: *krypton*. Th actual Python package name is **kr**.

The project contains a verification framework, mainly for signal processing (dsp) FPGA cores. This package uses MyHDL to implement the cores and the MyHDL simlution engine. Initially, the project is reimplementing the GnuRadio USRP HDL completely in Python/MyHDL. But the project is going slightly further and will make each of the DSP modules as generic and parameterizable as possible. It is the goal that the project will grow and support many different FPGA DSP systems.

The *krypton* project (kr or maybe kool-radio?) has the following sub-packages.

- *blocks* : This is a signal processing test/verification framework. It has a bunch of common *blocks* used in signal processing. This includes, sources, sinks, etc. This sub-package is used to test and verify the DSP components.
- *signal*: This package will look and feel like the scipy.signal package. The scipy.signal package is not used because this projects desires to run on *pypy* (very important!). This project cannot rely on the scipy packages. Common functions required will be port to pure Python implementations.
- *cores* : Contains the generic FPGA modules.
- *test* : Contains functional and performance tests for the previous sub-packages.
- *tools* : Python scripts for driving
- *scripts* : A set of useful scripts, example filter design GUIs etc.
- *examples* : Example projects build using the **kr** cores. This is really a sub-package but a directory, outside the kr package.

As of 3-Jan-2013 this project is just beginning and is in its infancy. Hopefully (with any luck) there will be significant progress in the following months (don't hold your breathe).

CHAPTER

TWO

INSTALLING

The krypton project is currently under development. There are no releases for the project (5-Jan-2013) and the project is under development.

USE AT YOUR OWN RISK

To use a current development snapshot the project needs to be checked out from the bitbucket repository using mercurial.

>> hg clone https://bitbucket.org/cfelton/krypton

After the project has been retrieved it can be installed into the Python environment:

>> python setup.py install

In the kr/test directory is a collection of tests that can be run with the nose test framework. Before running the tests see the next section and make sure dependencies are installed.

>> cd kr/test >> nosetests

As mentioned in the intro, most of the tests are intended to be run with pypy (else they will be really slow). The above *quick test* doesn't address setting up an environment that can run pypy and cpython. Ideally, there would be a small set of tests that would run in a reasonable amount of time with cpython that can be run.

CHAPTER

THREE

DEPENDENCIES

The following is a list of dependencies for the krypton project.

- MyHDL. MyHDL is used for the core simulation engine.
- nose. The nose.tools are used for verification.

Optional (kr.blocks verification framework and kr.signal tools)

- networkx. Networkx is used to create graphics of the system created.
- matplotlib. Used to generate plots.

CHAPTER

FOUR

WHAT IS BLOCKS

Blocks is a signal processing (digital signal processing) simulation and verification environment. Its main goal is to be used as a verification environment for complex digital systems. **Blocks** uses MyHDL as the core simulation engine. **Blocks** will be used to verify the *cores* Majority of the DSP tests and verification for the *cores* will use **Blocks**.

The intent of the sub-package is to create a convenient method of connecting DSP modules (blocks) with existing sinks, sources, and other *blocks*. This project has two parts, first the main block module and some factory functions for managing the *blocks*. Second, the collection of *blocks*, which are used to model, test, and verify the systems.

4.1 Basic Example

The following is a simple example how the **Blocks** package would be used.

```
import myhdl
from myhdl import Simulation

import kr
from kr.blocks.sources import Constant
from kr.blocks.sources import Sine
from kr.blocks.sinks import VerifySequence

c1 = Constant(1)
sine1 = Sine(frequency=40e3, sample_rate=1e6)
a1 = c1 + sine1
a2 = c1 + sine1
c3 = VerifySequence(a1,a2, val_list=[2,2,2,2,2,2], exact=True)

g = kr.blocks.GetAllGenerators()
Simulation(g).run()
```

The above example creates a simple system which has two sources (c1 and sine1) and two adders (a1 and a2) and one sink (c3). The *c3* block, *VerifySequence*, is used to verify the the expected output. After creating a system two factory functions are called, *blocks.GetAllGenerators* and *myhdl.Simulation*. These functions are used to run the simulation. There are other factory functions that are documented in the manual (one day I suppose).

4.2 Another Example

```
import myhdl
import kr
```

```
# ----[ Sources ]----
x = kr.blocks.sources.sine(
     frequency=1.234, sample_rate=100,
     phase_offset=0, amplitude=0.5, offset=0)

y = kr.blocks.sources.sine(
     frequency=3.333, sample_rate=100,
     phase_offset=0, amplitude=0.5, offset=0)

#----[ Processing Block ]----
z = x + y  #kr.blocks.add(x,y)

#----[ Sinks ]----
e    = kr.blocks.sinks.printx(z)
plt1 = kr.blocks.sinks.plotter(z, snapshots=True)

g = kr.blocks.GetAllGenerators()
myhdl.Simulation(g).run()
```

This example creates two sine sources and adds them together. The plotter can be used to create plots.

4.2.1 Current List of Blocks

- **Sources**
  - Constant
  - Sine
- **Sinks**
  - VerifySequence
  - Print
  - Plotter
  - VcdTracer
- **Basic**
  - Add (+)

4.3 Creating a Block

The following is the code for the *Constant* block. It shows what the bare minimal is to implement a block.

```
class Constant(Block):
_default_parameters = {
    'frequency' : 1e6
    }
_expected_num_of_inputs = (1,1)
_output_ports = ('inherit',)

def _initialize(self):
    # Use the strober block to generate the data valid pulses
    self.strober = Strober(clock=self.clock, reset=self.reset,
                           frequency=self.frequency)
```

```
        self.fs = self.strober.GetOutputSignals().strobe

    def _process(self):
        @instance
        def blk_process():
            while True:
                yield self.clock.posedge
                if self.fs:
                    for inp,outp in zip(self._inputs, self._outputs):
                        outp.signal.next = inp.signal
                        outp.strobe.next = True
                else:
                    for outp in self._outputs:
                        outp.strobe.next = False

        return blk_process



* _default_parameters : This is a class attribute dictionary that defines
                        the keyword parameters that can be set when
                        instantiating the block.  These are copied to the
                        instance attributes on *init*.  Each parameter is
                        avaialbe via an instance attribute.  Example,
                        c1 = Constant(1); c1.frequency.

  In this example, *frequency*, is the rate that the sources provide samples
  (sample rate).

* _expected_num_of_inputs : This indicates the number of inputs that can be
                            an input to the block

* _initialize : This is called before the simulation starts.

* _process : This is called during the simulation.  The *ports* that
  connect the blocks contain a strobe signal and a data signal.  The
  strobe indicates when the data is valid.  This is required in
  multi-rate systems.
```

4.3.1 Limitations

Don't use *numpy* and *scipy* functions. We want the simulations to be run with pypy. This limits the use of many third party packages available. In many cases this isn't an issue in some it can be. The *numpy* and *scipy* are really working on arrays and the simulation will work on one sample at a time. In some cases even though only a sample at a time is collect many *blocks* might buffer and want to do array/matrix operations. At this time avoiding using *numpy* and *scipy* is a must.

CHAPTER

FIVE

REGRESSION TESTING

@todo:

CHAPTER

SIX

MODULE DOCUMENTATION

For each block documentation is needed. Need to document what the blocks does and the parameters of the block. The following is the list of documentation for each the existing blocks.

6.1 kr.blocks

6.2 kr.blocks.sinks

6.3 kr.blocks.sources

CHAPTER

SEVEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*