rhea Documentation

Release 0.1pre

Christopher L. Felton

Dec 23, 2018

Contents

Note:

```
This project is in beta mode and under heavy development.  Not all
features described in the documentation are implemented.  The features
described in the documentation are in the process of being implemented.
Many of the modules, interfaces, and functions may change considerably
before the first release.
```

The `rhea` package is a collection of HDL cores written in myhdl. The `rhea` package is more than just a collection of cores it is also a framework for creating complex digital circuits. The `rhea` package includes a complete test suite.

The `rhea` package is divided into the following subpackages:

- **rhea**: The top-level namespace contains a small number of components (thin layer) use to build the subblocks (cores)
- **system**: The *system* subpackage contains the interface classes and other useful tools to assist in the building of complex digital designs.
- **models**: This subpackage contains various models used for development and verification
- **cores**: The *cores* subpackage contains the HDL implementation of the digital hardware cores.
- **build**: This subpackage automates various tool-flows (compilation). The automation mainly supports FPGA vendor tool-flows. The `build.boards` is a collection of board definitions. The build automation is used by selecting a board and automating the build for the board.
- **vendor***: The vendor subpackage is an encapsulation of device primitives.

CHAPTER 1

rhea common components

The `rhea` top-level namespace includes a small collection of functions and objects that are commonly used in building subblocks (cores). This is a thin-layer in the software stack used to build complex digital systems.

The following is the list of the functions and objects in the rhea top-level namespace. See the information below for more details.

1. rhea.Clock(init_val, frequency)
2. rhea.Reset(init_val, active, isasync)
3. rhea.Global()
4. rhea.Constants(**named_constants)
5. rhea.Signals(sigtype, num_sigs)
6. rhea.syncro
7. rhea.assign

class `rhea.`**`Clock`**(*val*, *frequency=1*)

class `rhea.`**`Reset`**(*val*, *active*, *isasync*)

class `rhea.`**`Global`**(*clock=None*, *reset=None*, *frequency=1*)

class `rhea.`**`Constants`**(***constargs*)

`rhea.`**`Signals`**(*sigtype*, *num_sigs*)

Create a list of signals :param sigtype: The type to create a Signal from. :type sigtype: bool, intbv :param num_sigs: The number of signals to create in the list :type num_sigs: int

Returns a list of signals all of type *sigtype*

Return type sigs

Creating multiple signals of the same type is common, this function helps facilitate the creation of multiple signals of the same type.

The following example creates two signals of bool

```
>>> enable, timeout = Signals(bool(0), 2)
```

The following creates a list-of-signals of 8-bit types

```
>>> mem = Signals(intbv(0)[8:], 256)
```

`rhea.`**`assign`**`()`

assign a = b

`rhea.`**`syncro`**`()`

signal synchronizer

Parameters

- **`sigin`** – signal input.
- **`sigout`** – synchronized signal output.
- **`posedge`** – a positive edge in the sync
- **`negedge`** – a negitive edge in the sync
- **`num_sync_ff`** – the number of sync stages.

1.1 Examples

CHAPTER 2

System

Note: System documentation is WIP and incomplete.

Need to take care of adding all the relevant classes and functions.

2.1 System overview

The `rhea` package provides frameworks for building complex digital circuits. These include modular and scalable interfaces and blocks. The following describes the specification for the frameworks being developed in the `rhea` package.

2.1.1 Control and status

One of the goals of the `rhea` package is to simplify the assembly of systems. In a complex digital system majority of the blocks will have two interfaces. One being the streaming data in and out of the module and the other a control and status interface. The control and status provides a lower-bandwidth interface into the component (block).

Defining a peripheral specific control-status object (CSO).

```
import rhea.system import ControlStatusBase

# define the control and status signals for a peripheral
class ControlStatus(ControlStatusBase):
    modes = enum("counting", "walking", "strobing")
    def __init__(self)
        self.enable = Signal(bool(0))
        self.pause = Signal(bool(0))
        self.mode = Signal(self.modes.counting)
```

In a peripheral either the default (defined) control-status object CSO can be used and added to the control-status interface.

```
@myhdl.block
def led_blinker(glbl, led, cso):
    # the cso interface provides the control and status for
    # this module
    assert isinstance(ControlStatus)
    clock, reset = glbl.clock, glbl.reset
    modes = ControlStatus.modes
    enabled = Signal(bool(0))

    @always_comb
    def beh_enabled():
        enabled.next = glbl.enable and cso.enable

    @always_seq(clock.posedge, reset=reset)
    def beh_blink():
        if enabled and not cso.pause:
            nextled = 0
            if cso.mode == modes.counting:
                # counting logic
            elif cso.mode == modes.walking:
                # walking logic
            elif cso.mode == modes.strobing
                # strobing logic

            led.next = nextled

    return beh_enable, beh_blink

led_blinker.cso = ControlStatus
```

The non-global control and status, i.e. the module specific control-status is accessed via the `cso`. This provides a clean encapsulation to the block (module). The `cso` can also include transactors to assist testing and verification.

Creating control status objects

The above example shows how a collections of control and status signals are defined in a class. To help guide the the tools, some additional information can be defined:

- `driven`: set the signal driven attribute to true to indicate a read-only (status) attribute.
- Use the hardware-types `Bit` and `Byte` to help drive the how the attributes are organized in a register-file. The `Bit` and `Byte` are only used to give hints to the register-file builder, if memory-mapped access is secondary use the standard **:myhdl:class:‘Signal‘**.
- Use `initial_value` property to overwrite the signals initial value, this is useful is static configurations.

2.1.2 Register files

When creating components for a design often a register file is included The register file is used for the control and status access (CSR) of the component. A register file is simply a collection of **‘registers‘_** that are used to control the component and read status. The register file is accessed by a memory-mapped bus. The register file provides dynamic control and status of the component.

The objects to create a register file encapsulate much of the detail required for typical register-file definition. In addition provides a mechanism for static definition (no bus present).

The following is a short example building a simple register file. Note the following is the manaul method to the example being used in this document. Utilizing the `ControlStatusBase` is an automated process, in majority of the cases register-files should not be explicitly defined but rather build from a CSO.

```
from rhea.system import RegisterFile, Register

# create a register file
regfile = RegisterFile()

# create a status register and add it to the register file
reg = Register('status', width=8, access='ro', default=0)
regfile.add_register(reg)

# create a control register with named bits and add
reg = Register('control', width=8, access='rw', default=1)
reg.add_named_bits('enable', bits=0, comment="enable the component")
reg.add_named_bits('pause', bits=1, comment="pause current operation")
reg.add_named_bits('mode', bits=(4, 2), comment="select mode")
regfile.add_register(reg)
```

Note: The current implementation requires all the register in a register file to be the same width.

The above example defines a register file to be used. This can be used in a new component/peripheral.

```
@myhdl.block
def led_blinker(glbl, membus, leds):
    clock = glbl.clock
    # instantiate the register interface module and add the
    # register file to the list of memory-spaces
    regfile.base_address = 0x8240
    regfile_inst = membus.add(glbl, regfile)

    # instantiate different LED blinking modules
    led_modules = (led_stroby, led_dance, led_count,)
    led_drivers = [Signal(leds.val) for _ in led_modules]
    mod_inst = []
    for ii, ledmod in enumerate(led_modules):
        mod_inst.append(ledmod(glbl, led_drivers[ii]))

    @always(clock.posedge)
    def beh_led_assign():
        leds.next = led_drivers[regfile.mode]

    return regfile_inst, mod_inst, beh_led_assign
```

The `led_blinker()` module demonstrates how to add the created `RegisterFile` to the memory-mapped bus and get a myhdl instance that provides the logic to read and write the register file from the bus interface passed to the module.

Note, in the above example a `base_address` was set. If the `base_address` attribute is not present the :class:‘MemoryMapped

2.1.3 Memory map interfaces

The :Register Files: section examples eluded to the memory-map (or CSR) interfaces and how they can be connected to register file. The `rhea` project contains the following memory-map interfaces:

- `Barebone`
- `Wishbone`
- `AvalonMM`
- `AXI4Lite`

Each of these implement a memory-map bus type/specification and each can be passed as and interface to a module. Each of the specific memory-mapped bus classes inherit the `MemoryMapped` class. The `MemoryMapped` defines the attributes and methods the memory-mapped buses have in common.

When interfacing to a register file, the register file is added to the bus as shown in the previous example with the `MemoryMapped.add()` function. The register file covers many use cases for adding control and status interfaces to different components. Each interface also contains a module to adapt the memory-map interface to a *generic* interface. In this case each bus is mapped to the `Barebone` bus with the `MemoryMapped.map_to_generic()` function / myhdl module.

The next section outlines how the `RegisterFile` and the corresponding registers is typically not used as defined above. Rather, an automated mapping of the control-status object is mapped to the memory-space. Software is used to encapsulate all the memory-based accesses.

2.1.4 From attributes to bus cycles

When designing a complex digital system with the `rhea` components we don't want to deal with creating explict memory-maps. We want to interface with various modules through their control-status attributes.

As defined in the above first example, for our simple LED blinker module there are a couple control signals defined. The module can be stimulated and controlled via this interface. We might have some external logic, or simply tie the module controls to physical inputs.

If we want to tie the controls to a register-file accessed by a memory-mapped this

```
@myhdl.block
def led_blinker(glbl, leds, membus=None, cso=None):

    if cso is None
        cso = led_blinker.cso()

    if membus is not None:
        rf = cso.get_register_file()
        membus.add(rf)

    # get any cso specific logic (if any)
    cso_inst = cso.get_generators()

    # ...
```

This gives a flexible mechanism to connect the module to a memory-mapped bus or simply control the module through some other mechanism (e.g. directly driven by the logic).

In the previous example all the explict addresses are hidden. The control-status attributes are accessed via the attributes (in simulation and host software) and all the memory-mapped bus accesses are hidden. The `MemoryMap` has utilities to export the memory-map.

Static configuration

The previous example demonstrated how the module can select to use the external `cso` object, default `cso`

2.2 Memory mapped interfaces

2.2.1 Base classes

The following are the building blocks for defining a system with memory-mapped attributes.

class `rhea.system.`**MemorySpace**

class `rhea.system.`**MemoryMapped**(*glbl=None*, *data_width=8*, *address_width=16*)

acktrans(*data=None*)
Acknowledge transaction

add = <myhdl._block._bound_function_wrapper object>

add_csr(*csr*, *name="*)

get_generic()
Get the generic bus interface Return the object that map_to_generic maps to. :return: generic bus interface

interconnect()
Connect all the components

map_from_generic(*generic*)
Map the generic bus (Barebone) to this bus This is a bus adapter that will adapt the generic bus to this bus. This is a module and returns myhdl generators

Parameters

- **generic** (*The generic memory-mapped bus, all the memory-mapped*) –
- **modules use the generic bus internally. This provides** (*supported*) –
- **agnostic bus interface to all the modules.** (*an*) –

Returns

Return type myhdl generators

map_to_generic(*generic*)
Map this bus to the generic (Barebone) bus This is a bus adapter, it will adapt the :return: generic bus, myhdl generators

peripheral_regfile(*regfile*, *name*, *base_address=0*)
override

Parameters

- **glbl** (*global signals, clock and reset*) –
- **regfile** (*register file interfacing to*) –

:param : :type : param glbl: global signals, clock and reset :param : :type : param regfile: register file interfacing to. :param : :type : param name: name of this interface :param : :type : param base_address: base address for this register file :param : :type : return: myhdl generators

readtrans(*addr*)
Read transaction

writetrans(*addr*, *data*)
Write transaction

class rhea.system.**MemoryMap**

class rhea.system.**RegisterFile**(*regdef=None*)

2.2.2 Memory mapped buses

The following are the memory-mapped bus interfaces available in rhea.

barebone

Note: Barebone bus needs additional documentation...

class rhea.system.memmap.**Barebone**(*glbl*, *data_width=8*, *address_width=8*, *name=None*, *num_peripherals=16*)

acktrans(*data=None*)
Acknowledge transaction

get_generic()
Get the generic bus interface Return the object that map_to_generic maps to. :return: generic bus interface

interconnect()

Returns

map_from_generic(*generic*)
In this case the *this* is the generic bus, use the signals passed, that is the expected behavior.

Parameters **generic** – the generic bus to map from

map_to_generic(*generic*)
In this case *this* is the generic bus, there is no mapping that needs to be done. Simply return ourself and all is good.

peripheral_regfile(*glbl*, *regfile*, *name*, *base_address=0*)
(arguments == ports) :param glbl: global signals, clock, reset, enable, etc. :param regfile: register file :param name: :param base_address:

readtrans(*addr*)
Read transaction

writetrans(*addr*, *data*)
Write transaction

wishbone

Note: Wishbone bus needs additional documentation...

class rhea.system.memmap.**Wishbone**(*glbl=None*, *data_width=8*, *address_width=16*, *name=None*)

acktrans(*data=None*)
acknowledge accessor for testbenches :param data: :return:

get_generic()
Get the generic bus interface Return the object that map_to_generic maps to. :return: generic bus interface

interconnect = <myhdl._block._bound_function_wrapper object>

peripheral_regfile = <myhdl._block._bound_function_wrapper object>

readtrans(*addr*)
read accessor for testbenches

writetrans(*addr*, *val*)
write accessor for testbenches Not convertible.

avalon

Note: Avalon bus needs additional documentation...

class rhea.system.memmap.**AvalonMM**(*glbl=None*, *data_width=8*, *address_width=16*, *name=None*)

acktrans(*data=None*)
Acknowledge transaction

interconnect = <myhdl._block._bound_function_wrapper object>

peripheral_regfile = <myhdl._block._bound_function_wrapper object>

readtrans(*addr*)
read accessor for testbenches :param addr: :return:

writetrans(*addr*, *val*)
write accessor for testbenches :param addr: address to write :param val: value to write to the address :return: yields

AXI4

Note: AXI4 bus needs additional documentation...

class rhea.system.memmap.**AXI4Lite**(*glbl*, *data_width=8*, *address_width=16*)

acktrans(*data=None*)
Acknowledge transaction

readtrans(*addr*)
Read transaction

writetrans(*addr*, *val*)
Emulate a write transfer from a master The following is a very basic write transaction, future enhancements are needed to verify/validate of features of the AXI4Lite bus.

@todo: add priority (not often used) @todo: add byte strobe @todo: add response checks @todo: and checks for all channel acks

2.3 register_file

Note: Needs added content

2.4 reference

Note: Needs added content

CHAPTER 3

Cores

The following is the user (and some developer) documentation on the various cores available in the *rhea* package.

3.1 First In, First Out (FIFO) cores

Various synchronous and asynchronous FIFO implementations.

3.1.1 fifo_sync

rhea.cores.fifo.**fifo_sync**()

Synchronous FIFO This block is a basic synchronous FIFO. In many cases it is better to use the *fifo_fast* synchronous FIFO (lower resources).

This FIFO uses a "read acknowledge", the read data is available on the read data bus before the read strobe is active. When the read signal is set it is acknowledging the data has been read and the next FIFO item will be available on the bus.

Parameters

- **glbl** (Global) – global signals, clock and reset
- **fbus** (*FIFOBus*) – FIFO bus interface
- **size** (*int*) – the size of the FIFO, the FIFO will have hold at maximum *size* elements.

Examples

Write and read timing:

```
clock:           /-\_/-\_/-\_/-\_/-\_/-\_/-\_/-\_/-\_/
fbus.write:      _/---\_______/-----------\___________
fbus.wrtie_data: -|D1 |-------|D2 |D3 |D4 |-----------
```

(continues on next page)

(continued from previous page)

```
fbus.read:       _____________/---\____________________
fbus.read_data:            |D1    |--------------------
fbus.empty:      ---------\______/--\__________________
```

Usage:

```
fifobus = FIFOBus(width=16)
fifo_inst = fifo_sync(glbl, fifobus, size=128)
```

3.1.2 fifo_async

rhea.cores.fifo.**fifo_async**()

The following is a general purpose, platform independent asynchronous FIFO (dual clock domains).

Cross-clock boundary FIFO, based on: “Simulation and Synthesis Techniques for Asynchronous FIFO Design”

Typically in the “rhea” package the FIFOBus interface is used to interface with the FIFOs

3.1.3 fifo_fast

rhea.cores.fifo.**fifo_fast**()

Often small simple, synchronous, FIFOs can be implemented with specialized hardware in an FPGA (e.g. vertically chaining LUTs).

This FIFO is intended to be used for small fast FIFOs. But when used for large …

This FIFO is a small FIFO (currently fixed to 16) that is implemented to take advantage of some hardware implementations.

Typical FPGA synthesis will infer shift-register-LUT (SRL) for small synchronous FIFOs. This FIFO is implemented generically, consult the synthesis and map reports.

Arguments (ports): glbl: global signals, clock and reset fbus: FIFOBus FIFO interface

Parameters **use_slr_prim** – this parameter indicates to use the SRL primitive (inferrable primitive). If SRL are not inferred from the generic description this option can be used. Note, srl_prim will only use a size (FIFO depth) of 16.

3.1.4 fifo_ramp

rhea.cores.fifo.**fifo_ramp**()

FIFO Ramp module This module provides a simple 8-bit counter that will generate a ramp. This ramp is fed to the USB fifo. This can be used to validate the usb connection and the device to host (IN) data rates.

3.1.5 fifo_tester

WIP

3.1.6 Examples

3.2 Serial Peripheral Interface (SPI)

The following is a description of the SPI cores in the `rhea` package.

3.2.1 SPI controller

`rhea.cores.spi.`**`spi_controller`**`()`

SPI (Serial Peripheral Interface) module This module is an SPI controller (master) and can be used to interface with various external SPI devices.

Parameters

- **`glbl`** (`Global`) – clock and reset interface
- **`spibus`** (*`SPIBus`*) – external (off-chip) SPI bus
- **`fifobus`** (*`FIFOBus`*) – interface to the FIFOs, write side is to the TX the read side from the RX.
- **`mmbus`** (`MemoryMapped`) – a memory-mapped bus used to access the control-status signals.
- **`cso`** (*`ControlStatus`*) – the control-status object used to control this peripheral
- **`include_fifo`** (*`bool`*) – include the FIFO . . . this is not fully implemented

Note: At last check the register-file automation was not complete, only the *cso* external control or *cso* configuration can be utilized.

3.2.2 SPI slave FIFO

`rhea.cores.spi.`**`spi_slave_fifo`**`()`

This is an SPI slave peripheral, when the master starts clocking any data in the TX FIFO (fifobus.write) will be sent (the next byte) and the received byte will be copied to RX FIFO (fifobus.read). The *cso* interface can be used to configure how the SPI slave peripheral behaves.

(Arguments == Ports) :param glbl: global clock and reset :type glbl: Global :param spibus: the external SPI interface :type spibus: SPIBus :param fifobus: the fifo interface :type fifobus: FIFOBus :param cso: the control status signals :type cso: ControlStatus

3.3 fpgalink

This is a MyHDL implementation of the HDL for the *fpgalink* project. The fpgalink HDL core can be instantiated into a design:

For simulation and verification the *fpgalink* interface can be stimulated using the FX2 model and high-level access functions:

The following is a pictorial of the verification environment .

For more information on the [fpgalink]() software, firmware, and general design information see [makestuff]().

3.4 usbp

USB Peripheral, this is another Cypress FX2 controller interface, this has two interfaces a “control” interface and a “streaming” interface. This FX2 interface is intended to work with the [fx2 firmware]() that configures the controller as a USB CDC/ACM device (virtual serial port). The [fx2 firmware]() also has a couple vendor unique commands that can be sent using the pyusb (or other low-level USB interfaces like libusb). The Python version of the host software (including firmware) can be retrieved via pip

```
>> pip install usbp
>>> import usbp
>>> import serial
```

One of the tricky items with USB devices is setting the permissions correctly. On a linux system to set the . . .

CHAPTER 4

Build

The *rhea* package includes tools to automate the vendor FPGA toolflows.

CHAPTER 5

Board definitions

The *rhea.build* contains a large list of board definitions. The board definitions define an FPGA and its configuration for a particular board. The board definitions define the default port list for the FPGA on a particular board. The following is a guideline on how to add a new board definition.

The information needed to create a board definition comes from the boards datasheet and/or schematic. To make it easy to trace back to the original documentation the port names should match the net names in the documentation / schematic with a few exceptions. The port names should be lowercase (this will be one difference from the documentation / schematic).

The board definitions are a subclass of the *FPGA* class. The FPGA information is captured in the class attributes.

5.1 Example: XESS CAT board

The following is a minimal example creating a board definition for the XESS CAT board. From the CAT board schematics the port definitions can be defined.

```
class CATBoard(FPGA):
    vendor = 'lattice'
    family = 'ice40'
    device = 'HX8K'
    packet = 'CT256'
    _name = 'catboard'

    default_clocks = {
        'clock': dict(freqeuncy=100e6, pins=('C8',))
    }

    default_ports = {
        'led': dict(pins=('A9', 'B8', 'A7', 'B7',)),
        'sw': dict(pins=('A16', 'B9',)),
        'dipsw': dict(pins=('C6', 'C5', 'C4', 'C3',)),
        'hdr1': dict(pins=('J1', 'K1', 'H1', 'J2', )),
    }
```

5.2 Extending definitions

In many situations the top-level module ports might not match the default ports in the board definition or the user might want to create a different board definition.

5.2.1 Mapping port names

There are two functions available in a specific board definition object: `add_port` and `add_port_name`. When the pin is known use `add_port` and when the default port name is known but a different name is desired use `add_port_name` to add a new port name that maps to the properties of an existing port. See the `pone` example for an `add_port_name` use.

5.2.2 Creating a custom board definition

Custom board defintions can be created from the standard board definitions contained in the *rhea.build.boards* collection. For example, a board might have many connectors or generic IO. The user could have a boards with specific hardware attached. In these cases the user many wish to create a new custom board definition.

```
class MyCustomBoard(Xula2):
    # overriding the default ports, inherit the default
    # clocks.  The default ports in this cause reprsent
    # the various widgets connected to the Xula2+stickit
    default_ports = {
        'leds': dict(pins=('R7', 'R15', 'R16', 'M15',)),
        'btns': dict(pins=('F1', 'F2', 'E1', 'E2',))
    }
```

Contents:

CHAPTER 6

Examples

The following are the examples available in the examples directory.

6.1 Xess Corp. Boards

- **Xula(2)**
  - binary hello (blinky)
  - VGA (TBC)
  - SDRAM (TBC)
- **CAT Board**
  - binary hello (blinky)
  - SDRAM (TBC)

6.2 Digilent Boards

- **Nexys**
  - **‘binary hello (blinky) <>‘_**
  - fpgalink
  - usbp (TBC)
- **Atlys**
  - **‘binary hello (blinky) <>‘_**
  - fpgalink (TBC)
  - usbp (TBC)

- **Zybo**
    - binary hello (blinky)

6.3 Terasic Boards

- **DE0nano**
    - binary hello (blinky)
    - LT24 LCD (colorbars)
    - ADC and Accelerometer (WIP)
    - SDRAM (TBC)

6.4 Lattice Boards

- **ICEStick**
    - binary hello (blinky)

6.5 Misc. Boards

- **Open-Source UFO-400**
    - **‘binary hello (blinky) <>‘_**
    - usbp (TBC)
- **DSPtronics Signa-X1 (sx1)**
    - **‘binary hello (blinky) <>‘_**
    - fpgalink (TBC)
    - usbp (TBC)
    - **audio examples**
        - audio echo (TBC)
        - audio streaming (TBC)

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Index