cocotbext busbase

April 4, 2025

Jay Convertino

Contents

1 Usage

1.1 Introduction

Cocotb extension to provide base of bus methods.

1.2 Dependencies

The following are the dependencies of the cores.

- iverilog (simulation)
- cocotb (simulation)
- cocotb-bus (simulation)

1.3 In a Simulation

Below is a simple example of creating a basic master for a bus using busmaster (see test.py for usable example).

```
# Class: basictrans
# create an object that associates a data member and
    ↪ address for operation.
class basictrans(transaction):
    def __init__(self, address, data=None):
        self.address = address
        self.data = data

# Class: basicMaster
# basic bus master
class basicMaster(busbase):
  # Variable: _signals
  # List of signals that are required
  _signals = ["addr", "we", "cs", "data"]

  # Constructor: __init__
  # Setup defaults and call base class constructor.
  def __init__(self, entity, name, clock, reset, *args,
      ↪ **kwargs):
    super().__init__(entity, name, clock, *args, **kwargs
        ↪ )

    self.log.info("BASIC␣Master")
    self.log.info("Copyright␣(c)␣2025␣Jay␣Convertino")
    self.log.info("https://github.com/johnathan-
        ↪ convertino-afrl/cocotbext-busbase")
```

```
    self._reset = reset

    self.bus.addr.setimmediatevalue(0)
    self.bus.data.setimmediatevalue(0)

  # Function: read
  # Read from a address and return data
  async def read(self, address):
    trans = None
    if(isinstance(address, list)):
      temp = []
      for a in address:
        temp.append(basictrans(a))
      temp = await self.read_trans(temp)
      #need a return with the data list only. This is
          ↪ only a guess at this point
      return [temp[i].data for i in range(len(temp))]
    else:
      trans = await self.read_trans(basictrans(address))
      return trans.data

  # Function: write
  # Write to a address some data
  async def write(self, address, data):
    if(isinstance(address, list) or isinstance(data, list
        ↪ )):
      if(len(address) != len(data)):
        self.log.error(f'Address␣and␣data␣vector␣must␣be␣
            ↪ the␣same␣length')
      temp = []
      for i in range(len(address)):
        temp.append(basictrans(address[i], data[i]))
      await self.write_trans(temp)
    else:
      await self.write_trans(basictrans(address, data))

  # Function: _check_type
  # Check and make sure we are only sending 2 bytes at a
      ↪ time and that it is a bytes/bytearray
  def _check_type(self, trans):
      if(not isinstance(trans, basictrans)):
          self.log.error(f'Transaction␣must␣be␣of␣type:␣{
              ↪ type(basictrans)}')
          return False
```

```
    return True

# Method: _run
# _run thread that deals with read and write.
async def _run(self):
  self.active = False

  trans = None

  while True:

    if self._reset.value:
      self.bus.we.setimmediatevalue(0)
      self.bus.cs.setimmediatevalue(0)
      await RisingEdge(self.clock)
      continue

    if not self.wqueue.empty():
      self.active = True
      while self.active:
        trans = await self.wqueue.get()
        self.bus.we.setimmediatevalue(1)
        self.bus.cs.setimmediatevalue(1)
        self.bus.addr.setimmediatevalue(trans.address)
        self.bus.data.setimmediatevalue(trans.data)
        self._idle_write.set()
        await RisingEdge(self.clock)

        self.active = not self.wqueue.empty()
    elif not self.qqueue.empty():
      self.active = True
      while self.active:
        trans = await self.qqueue.get()
        self.bus.we.setimmediatevalue(0)
        self.bus.cs.setimmediatevalue(1)
        self.bus.addr.setimmediatevalue(trans.address)
        trans.data = self.bus.data.value
        await self.rqueue.put(trans)
        self._idle_read.set()
        await RisingEdge(self.clock)

        self.active = not self.qqueue.empty()
    else:
      self.active = False
      self.bus.we.setimmediatevalue(0)
      self.bus.cs.setimmediatevalue(0)
```

```
        await RisingEdge(self.clock)
```

2 Architecture

Please see 4 for more information.

busbase is the base class for busbase methods.

noSignal used if a signal does not exist, you can switch it out with this class and still use the value attribute.

transaction is the base class for all transactions. This will contain items such as address/data for read/write.

2.1 Directory Guide

Below highlights important folders from the root of the directory.

1. **docs** Contains all documentation related to this project.
    - **manual** Contains user manual and github page that are generated from the latex sources.
2. **cocotbext** Contains source files for the extension
    - **busbase** Contains source files for busbase
3. **tests** Contains test files for cocotb

3 Simulation

A simulation for testing the cores can be run to verify operation.

3.1 cocotb

To use the cocotb tests you must install the following python libraries.

```
$ pip install cocotb
$ pip install -e .
```

Then you must enter the tests folder and enter the folder of the type you wish to test. From there you may execute the following command which will kick off the test.

```
$ make
```

4 Code Documentation

Natural docs is used to generate documentation for this project. The next lists the following sections.

- **init** Python init code.
- **busbase** Contains bus base for threads and read/write methods.
- **cocotb test** Python TestFactory code.
- **cocotb verilog test wrapper** Verilog wrapper module.

__init__.py

AUTHORS

JAY CONVERTINO

DATES

2025/03/26

INFORMATION

Brief

uP define for packages

License MIT

busbase.py

AUTHORS

JAY CONVERTINO

DATES

2025/03/11

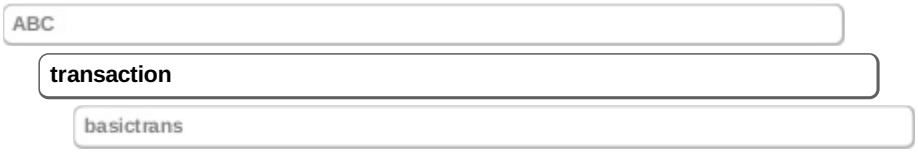
INFORMATION

Brief

classic bus define for packages

License MIT

transaction



Abstract class for transaction types

noSignal

noSignal

Class to use when a signal does not exist

busbase

busbase

basicMaster

A busbase to transmit test routine.

FUNCTIONS

__init__

```
def __init__(
self,
entity
:
SimHandleBase,
name
:
Optional[str],
clock
:
SimHandleBase,
*
args
:
Any,
**
kwargs
:
Any
)
```

Initialize the object

VARIABLES

wqueue

```
self.wqueue
```

Queue to store write requests

qqueue

```
self.qqueue
```

Queue to store read requests

rqueue

```
self.rqueue
```

Queue to store result of read requests

_idle_read

```
self._idle_read
```

Event trigger for cocotb read

_idle_write

```
self._idle_write
```

Event trigger for cocotb write

self._run_cr

```
self._run_cr
```

Thread instance of _run method

FUNCTIONS

_restart

```
def _restart(
self
)
```

kill and restart _run thread.

write_count

```
def write_count(
self
)
```

How many items in the write queue

read_count

```
def read_count(
self
)
```

How many items in the read queue

write_empty

```
def write_empty(
self
)
```

Is the quene empty?

read_empty

```
def read_empty(
self
)
```

Is the quene empty?self.bus.penable.value

write_clear

```
def write_clear(
self
)
```

Remove all write items from queue

read_clear

```
def read_clear(
self
)
```

Remove all read items from queue

wait_read

```
async def wait_read(
self
)
```

Wait for the run thread to become idle from a read.

wait_write

```
async def wait_write(
self
```

```
)
```

Wait for the run thread to become idle from a write.

idle

```
def idle(
self
)
```

Are all the queues empty and the _run is not active processing data.

write_trans

```
async def write_trans(
self,
trans
:
transaction
)
```

Write transaction to send to write queue

read_trans

```
async def read_trans(
self,
trans
:
transaction
)
```

Read bus and output and tranaction.

_write

```
async def _write(
self,
trans
:
transaction
)
```

Write data one element at a time

_queue_read

```
async def _queue_read(
self,
trans
:
transaction
)
```

Setup queue for read requests

_read

```
async def _read(
self,
trans
:
transaction
)
```

Read dat one element at a time

_check_type

```
def _check_type(
self,
trans
)
```

Check and make sure we are only sending the correct transaction type

_run

```
async def _run(
self
)
```

Virtual method for _run thread that deals with read and write queues.

basictrans

create an object that associates a data member and address for operation.

basicMaster

basic bus master

VARIABLES

_signals

```
_signals
```

List of signals that are required

FUNCTIONS

__init__

```
def __init__(
self,
entity,
name,
clock,
reset,
                                                                *
args,
                                                               **
kwargs
)
```

Setup defaults and call base class constructor.

read

```
async def read(
self,
address
)
```

Read from a address and return data

write

```
async def write(
self,
address,
data
)
```

Write to a address some data

_check_type

```
def _check_type(
self,
trans
)
```

Check and make sure we are only sending 2 bytes at a time and that it is a bytes/bytearray

_run

```
async def _run(
self
)
```

_run thread that deals with read and write.

basicEchoSlave

Respond to master reads and write by returning data, simple echo core.

VARIABLES

_signals

```
_signals
```

List of signals that are required

FUNCTIONS

__init__

```
def __init__(
self,
```

```
 entity,
 name,
 clock,
 reset,
 numreg
 =
 256,
                                                                    *
 args,
                                                                   **
 kwargs
)
```

Setup defaults and call base class constructor.

_check_type

```
def _check_type(
self,
trans
)
```

Check and make sure we are only sending 2 bytes at a time and that it is a bytes/bytearray

_run

```
async def _run(
self
)
```

_run thread that deals with read and write.

TB

TB

Create the device under test which is the master/slave.

FUNCTIONS

run_test

```
async def run_test(
dut,
payload_data
 =
None
)
```

Tests the source/sink for valid transmission of data.

incrementing_payload

```
def incrementing_payload()
```

Generate a list of ints that increment from 0 to 2^8

test

```
def test(
request
)
```

Main cocotb function that specifies how to put the test together.

test.v

AUTHORS

JAY CONVERTINO

DATES

2025/04/04

INFORMATION

Brief

Test bench for busbase

License MIT

test

```
module test #(
parameter
ADDRESS_WIDTH
  =
32,
parameter
BUS_WIDTH
  =
4
) ( input clk, input rst, inout [ADDRESS_WIDTH-1:0] b_addr, inout b_we, inou
```

Test bench loop busbase example

Parameters

ADDRESS_WIDTH parameter	Width of the address in bits.
BUS_WIDTH parameter	Width of the data in bytes.

Ports

clk	Clock for all devices in the core
rst	Negative reset
b_addr	address to read write from
b_we	write enable 1, read 0
b_cs	chip select, 1 selected
b_data	data input/output