basil Documentation

Release 3.2.1dev0

T

Jun 25, 2021

CONTENTS

Basil is a modular readout framework intended to allow simple and fast data acquisition systems (DAQ) design. It consists of different hardware components, FPGA firmware modulus and a Python based contol software.

Contents:

CHAPTER

ONE

INTRODUCTION

Basil is a modular readout framework intended to allow simple and fast data acquisition systems (DAQ) design. It consists of different hardware components, FPGA firmware modulus and a Python based control software.

1.1 Features

Firmware:

- very simple single master bus definition
- multiple basic modules (ex. SPI, SEQ)
- multiple interfaces (UART, USB2, USB3, Ethernet)

Software:

- layer structure following hardware
- generation based on yaml file
- register abstract layer (RAL)
- simulator interface allows software test against simulated RTL (thanks to cocotb)

1.2 Installation

From host folder run:

```
python setup.py install
```

or

```
pip install -e "git+https://github.com/SiLab-Bonn/basil.git#egg=basil&subdirectory=host"
```

1.3 Simulation

Thank to Chris Higgs basil has a simulation interface (SiSim) with allow communication with simulator as if talking to real hardware.

To make simulation one need:

- verilog simulator (ex. Icarus)
- cocotb library
- set interface type to SiSim

Basil unit tests make extensive use of this feature. See tests folder.

1.4 License

If not stated otherwise.

Host Software: The host software is distributed under the BSD 3-Clause ("BSD New" or "BSD Simplified") License.

FPGA Firmware: The FPGA software is distributed under the GNU Lesser General Public License, version 3.0 (LGPLv3).

CHAPTER

TWO

HARDWARE

Basil also allows easy integration with custom Hardware.

For our purpuse we developed general purpose hardware components which connect to custom DUTs. The MultiIO board (MIO) implements the main FPGA for digital IO, memory ressources, and the interface (USB 2.0) to the PC. To add analog functionality, the Gerneral Purpose Analog Card (GPAC) was developed. It connects to the MultiIO board and provides 4 programmable power supplies, 12 current sources, 4 voltage sources, 4 fast ADC channels, digital IO with scalable voltages, LVDS, and a programmable injection pulse generator. The device under test (DUT) will be connected with a custom PCB to the MIO/GPAC hardware.

- MultiIO - Digital IO card with FPGA and USB 2.0 interface
- MIO3 - Digital IO card with FPGA and USB 3.0 interface based on Enclustra KX1 module
- GPAC - General Purpose Analog Card

2.1 MIO (Multi IO Card)

The “S3 Multi IO System” is developed as an easy to use multi purpose digital IO card. It includes a free programmable Xilinx Spartan3 FPGA, SRAM Memory, USB2.0 Interface and a 8051 microcontroller with I2C and SPI functionality. It is designed to provide sufficient digital IO capability to any kind of daughter card.

Features:

Silicon devices

- Xilinx Spartan3 FPGA - XC3S1000 FG320 4C
- Cypress USB Controller - CY7C68013A 128AXC
- Cypress async. SRAM - CY7C1061AV33 10ZXC
- Programmable clock generator - Cypress CY22150

IO connections

- USB2.0 B-type as host interface
- Multi-IO-Connector with 80 user IO´s (VccIO 1:2 V to 3:3 V)
- Agilent debug connector (1253-3620)
- JTAG connection
- RJ-45 connector for 2 LVDS transmitter and 2 LVDS receiver
- Header with I2C and SPI functionality
- Header with additional FPGA user IO´s
- 3 buffered LVTTL outputs with LEMO
- 3 buffered LVTTL inputs with LEMO

Power supply

- via external 5V supply
- via USB cable

Configuration capability

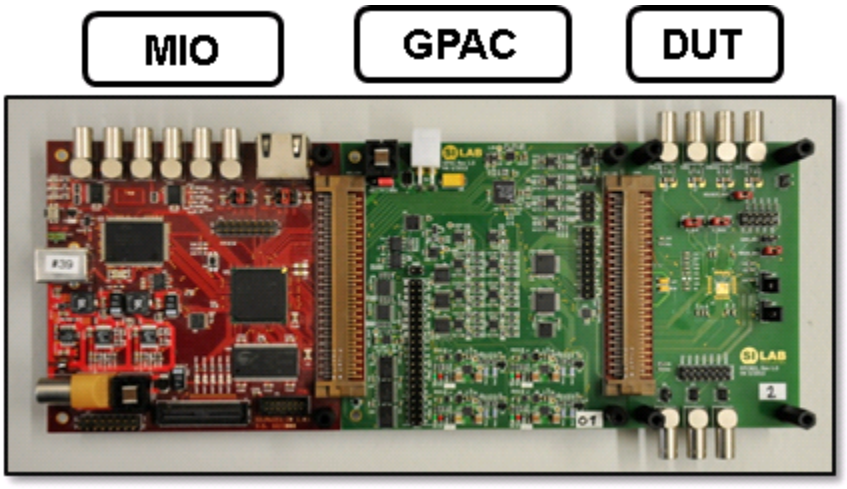
- via JTAG

- via USB2.0

Documentation for MIO card.

2.2 GPAC (General Purpose Analog Card)

GPAC Card is developed as an easy to use multi purpose analog IO card compatible with MIO Card.



Features:

- 4 regulated power supples, 0.8-1.83/2.83 V, max. 1000 mA, (controlled by I2C)
- 4 RX and 4 TX LVDS Lines
- 4 channel ADC, 25MS, 14bit
- 16 CMOS Outputs
- 8 CMOS Inouts
- 12 current source/sink, -1mA to +1mA, 12bit (controlled by I2C)
- 4 voltage outputs, 0-2.048 V, 12bit (controlled by I2C)
- 64x (4 available to DUT) channel slow ADC for monitoring (controlled by I2C)
- Injection Pulse Generator with programmable voltage levels (high and low)

Documentation for GPAC card.

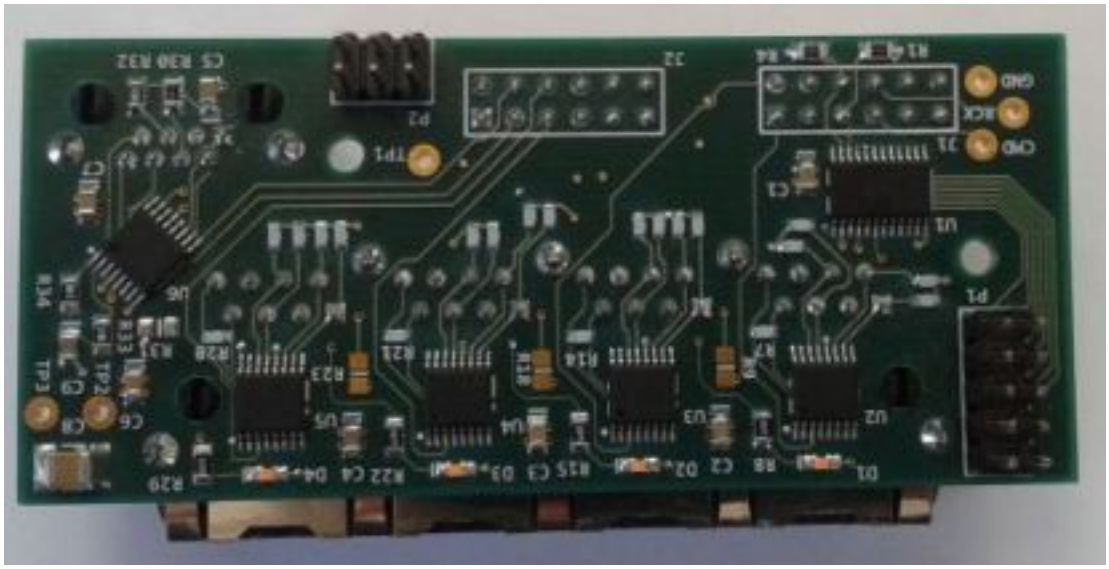
2.3 MIO3 (Multi IO Card USB3)

TBD.

2.4 LX9

LX9 Board.

4 channel FE-I4 adepter with TLU:



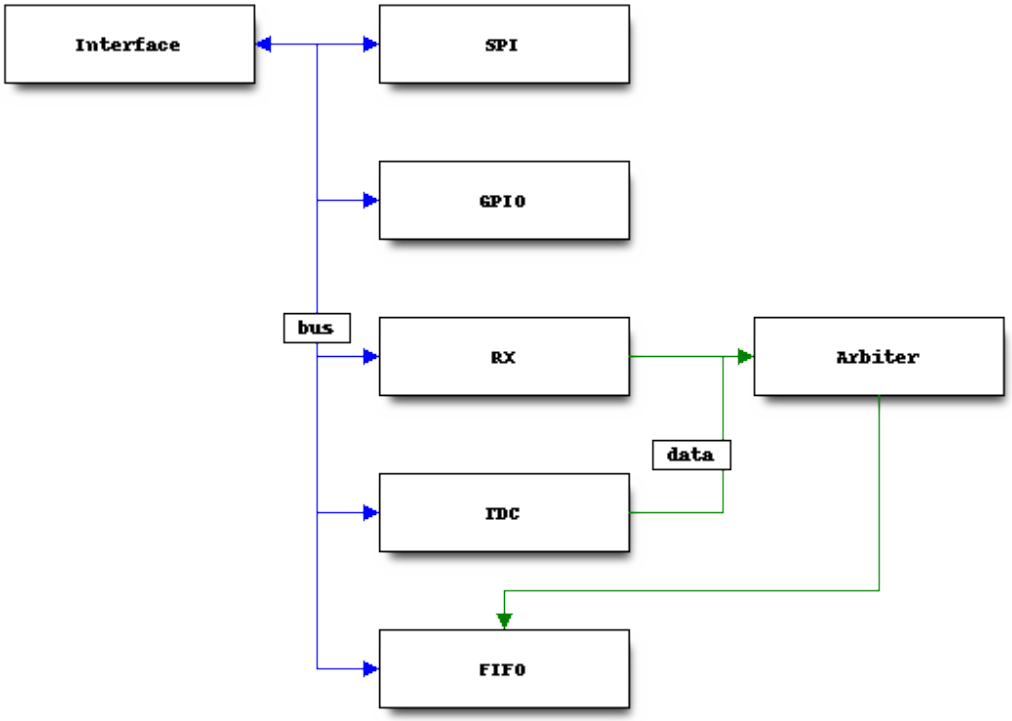
CHAPTER

THREE

FIRMWARE

FPGA firmware consists of very simple single master bus definition and set of standard modules used by DAQ systems.

Typical firmware consists of basil bus connecting all modules. Control modules witch provide configuration to DUT (like SPI/GPIO) and data taking modules (like data receivers). Received data (32 bit) are stored in the FIFO (large extremal memory) and can be continuously pulled from host application. Data from different modules are identified by source codding in 32bit data words.



3.1 basil bus

single write

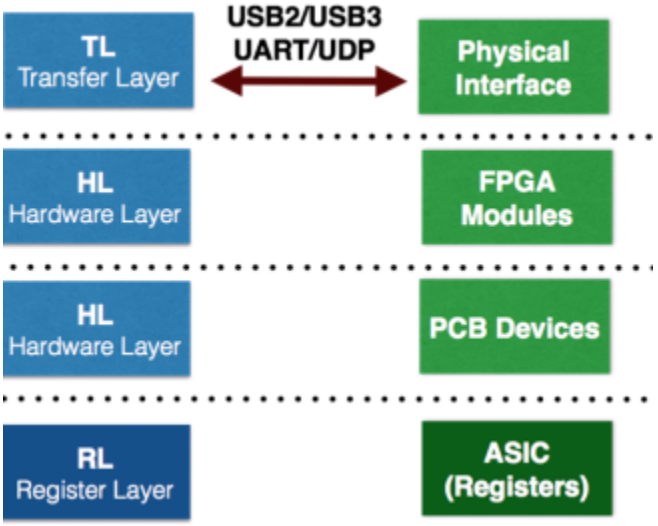
single read

CHAPTER

FOUR

SOFTWARE

The software framework has a modular structure that reflects the firmware and adds extra layers to make hardware interface user friendly. It loosely follows Register Abstract Layer (RAL) concepts. All the layers are automatically created based on yaml configuration file.



4.1 Yaml configuration file

TBD

4.2 Transfer Layer (TL)

Implements communication interface like UART, USB, Ethernet or Simulation. Every TL interface implements 2 functions:

class basil.TL.TransferLayer.**TransferLayer**(*conf*)

Transfer Layer implements minimum API needed access to hardware. On error `raise IOError`.

init()

Initialize and connect to hardware.

read()

Read access.

Return type None

write(*data*)
Write access.

Parameters **data** (*iterable*) – array/list of bytes

Return type None

4.3 Hardware Layer (HL)

Implements drivers for basil modules and external devices. Drivers can reference other drivers within HL.

4.4 Register Layer (RL)

Implements Register Level Abstraction. Allow to user/control software to work on DUT registers without taking thinking about underlying levels.

CHAPTER

FIVE

MODULES

5.1 Driver

class basil.HL.gpio.**gpio**(*intf*, *conf*)

GPIO interface

reset()

Soft reset the module.

5.2 Driver

class basil.HL.spi.**spi**(*intf*, *conf*)

Implement serial programming interface (SPI) driver.

get_data(*size=None*, *addr=None*)

Gets data for incoming stream

get_en()

Gets state of enable.

get_repeat()

Gets Number of repetitions of sequence with delay 'wait' (if 0 –> repeat forever)

get_size()

Get size of shift register length

get_wait()

Gets time delay between repetitions in clock cycles

is_done()

Get the status of transfer/sequence.

reset()

Soft reset the module.

set_data(*data*, *addr=0*)

Sets data for outgoing stream

set_en(*value*)

Enable start on external EXT_START signal (inside FPGA)

set_repeat(*value*)

If 0: Repeat sequence forever Other: Number of repetitions of sequence with delay 'wait'

set_size(*value*)
Number of clock cycles for shifting in data ex. length of matrix shift register (number of pixels daisy chained)

set_wait(*value*)
Sets time delay between repetitions in clock cycles

start()
Starts the shifting data

5.3 Driver

class basil.HL.seq_gen.**seq_gen**(*intf*, *conf*)
Sequencer generator controller interface for seq_gen FPGA module.

5.4 Driver

class basil.HL.pulse_gen.**pulse_gen**(*intf*, *conf*)
Pulser generator

get_en()
Return info if pulse starts with a fixed delay w.r.t. shift register finish signal (true) or if it only starts with .start() (false)

set_delay(*value*)
Pulse delay w.r.t. shift register finish signal [in clock cycles(?)]

set_en(*value*)
If true: The pulse comes with a fixed delay with respect to the external trigger (EXT_START). If false: The pulse comes only at software start.

set_repeat(*value*)
Pulse repetition in range of 0-255

set_width(*value*)
Pulse width in terms of clock cycles

start()
Software start of pulse at random time

5.5 Driver

class basil.HL.seq_rec.**seq_rec**(*intf*, *conf*)
Sequencer receiver controller interface for seq_rec FPGA module.

5.6 Driver

class basil.HL.cmd_seq.**cmd_seq**(*intf*, *conf*)
FEI4 Command Sequencer Controller Interface for cmd_seq FPGA module.

5.7 Driver

class basil.HL.fei4_rx.**fei4_rx**(*intf*, *conf*)
FEI4 receiver controller interface for fei4_rx FPGA module

5.8 Driver

class basil.HL.fast_spi_rx.**fast_spi_rx**(*intf*, *conf*)
Fast SPI interface

reset()
Soft reset the module.

5.9 Driver

class basil.HL.tlu.**tlu**(*intf*, *conf*)
TLU controller interface

5.10 Driver

class basil.HL.tdc_s3.**tdc_s3**(*intf*, *conf*)
TDC controller interface

5.11 Driver

class basil.HL.sram_fifo.**sram_fifo**(*intf*, *conf*)
SRAM FIFO controller interface for sram_fifo FPGA module.

property FIFO_INT_SIZE
Get FIFO size in units of integers (32 bit).

fifo_size [int] FIFO size in units of integers (32 bit).

get_FIFO_INT_SIZE()
Get FIFO size in units of integers (32 bit).

fifo_size [int] FIFO size in units of integers (32 bit).

get_data()
Reading data in SRAM.

array [numpy.ndarray] Array of unsigned integers (32 bit).

get_fifo_int_size()
Deprecated Get FIFO size in units of integers (32 bit).

fifo_size [int] FIFO size in units of integers (32 bit).

get_fifo_size()
Deprecated Get FIFO size in units of bytes (8 bit).

fifo_size [int] FIFO size in units of bytes (8 bit).

get_read_error_counter()
Deprecated Get read error counter.

fifo_size [int] Read error counter (read attempts when SRAM is empty).

get_size()
Deprecated

5.12 Driver

class basil.HL.fadc_rx.**fadc_rx**(*intf*, *conf*)
Fast ADC channel receiver

set_align_to_sync(*value*)
Align data taking to a synchronization signal, reset signal is the synchronization signal (hard coded connection in Verilog source code)

set_single_data(*value*)

5.13 Driver

class basil.HL.i2c.**i2c**(*intf*, *conf*)
Implement master i2c programming interface driver.

property is_ready

raises ExceptionType IOError

Transfer not acknowledged.

read(*addr*, *size*)
Read access.

Parameters

- **addr** (*char*) – i2c slave address
- **size** (*int*) – size of transfer

Returns data byte array

Return type array.array('B')

write(*addr*, *data*)
Write access.

Parameters

- **addr** (*char*) – i2c slave address
- **data** (*iterable*) – array/list of bytes

Return type None

CHAPTER

SIX

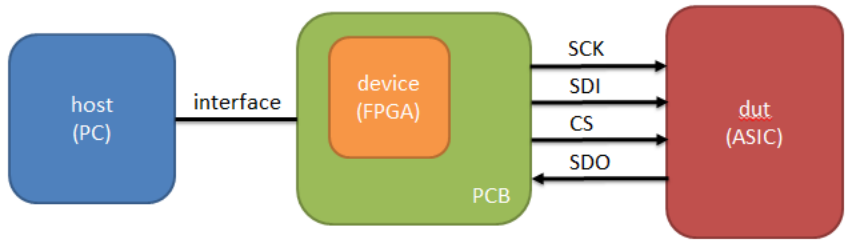
EXAMPLES

Example of project can be found in examples folder.

For more usecases check also tests folder.

6.1 spi

An example shows how to create a simple spi interface.



Instantiate a verilog spi module in the firmware verilog code.

```
localparam SPI_BASEADDR = 32'h1000;
localparam SPI_HIGHADDR = 32'h1FFF;

spi
#(
    .BASEADDR(SPI_BASEADDR),
    .HIGHADDR(SPI_HIGHADDR),
    .MEM_BYTES(4)
)  i_spi
(
    .BUS_CLK(BUS_CLK),
    .BUS_RST(BUS_RST),
    .BUS_ADD(BUS_ADD),
    .BUS_DATA(BUS_DATA),
    .BUS_RD(BUS_RD),
    .BUS_WR(BUS_WR),

    .SPI_CLK(SPI_CLK),

    .SCLK(SCLK),
    .SDI(SDI),
```

(continues on next page)

(continued from previous page)

```
    .SDO(SDO),
    .SEN(SEN),
    .SLD(SLD)
);
```

Create a configuration file.

```
transfer_layer:
  - name  : intf
    type  : SiSim

hw_drivers:
  - name      : spi
    type      : spi
    interface : intf
    base_addr : 0x1000
    mem_bytes : 2

  - name        : CNT
    type        : StdRegister
    hw_driver   : spi
    size        : 16
    fields:
      - name    : EN
        size    : 1
        offset  : 15
      - name    : OUT
        size    : 15
        offset  : 14
```

Write control program.

```
dut = Dut('spi.yaml')
dut.init()

dut['CNT']['EN'] = 1
dut['CNT']['OUT'] = 0x00f0
dut['CNT'].write()
dut['CNT'].start()

while not dut['CNT'].is_ready():
    pass
```

Result of simulation:

A workin example can be seen in tests/test_SimSpi.py.

6.2 gpio

TBD.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

INDEX

P

R

S

T

W