High level Firmware Design with **ARGG_HDL**

Python based Hardware Description Language (Library)

Richard Peschke | rp40@hawaii.edu

Introduction

High Level Programming for Firmware Design

- Clean Code through Objects
- Generic Code through Templates
  and customizable "HDL Converters"
- Powerful Tools by using an existing Language

Clean Code through Objects

Clean Code through Objects

Object-Oriented Design

Encapsulation

Abstraction

Polymorphism

Class

Information Hiding

Inheritance

Object-Oriented Design:

- is fundamental for (virtually) all High-Level Programming Languages
- Is very successful when modeling complex system

Encapsulation

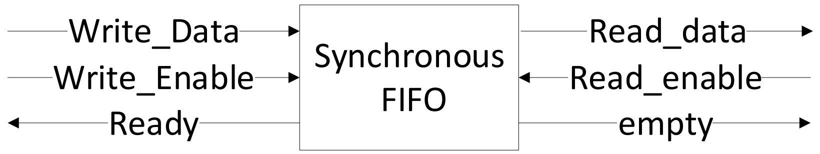
- An Object has one task and it contains all components it needs to fulfill this task
- Objects are modified through function
  ➔Preserving invariance
  ➔Avoiding Undefined behavior

Information Hiding

- Objects are defined by there public interface not by their internal structure
- An objects internal structure is not important for the user of the object

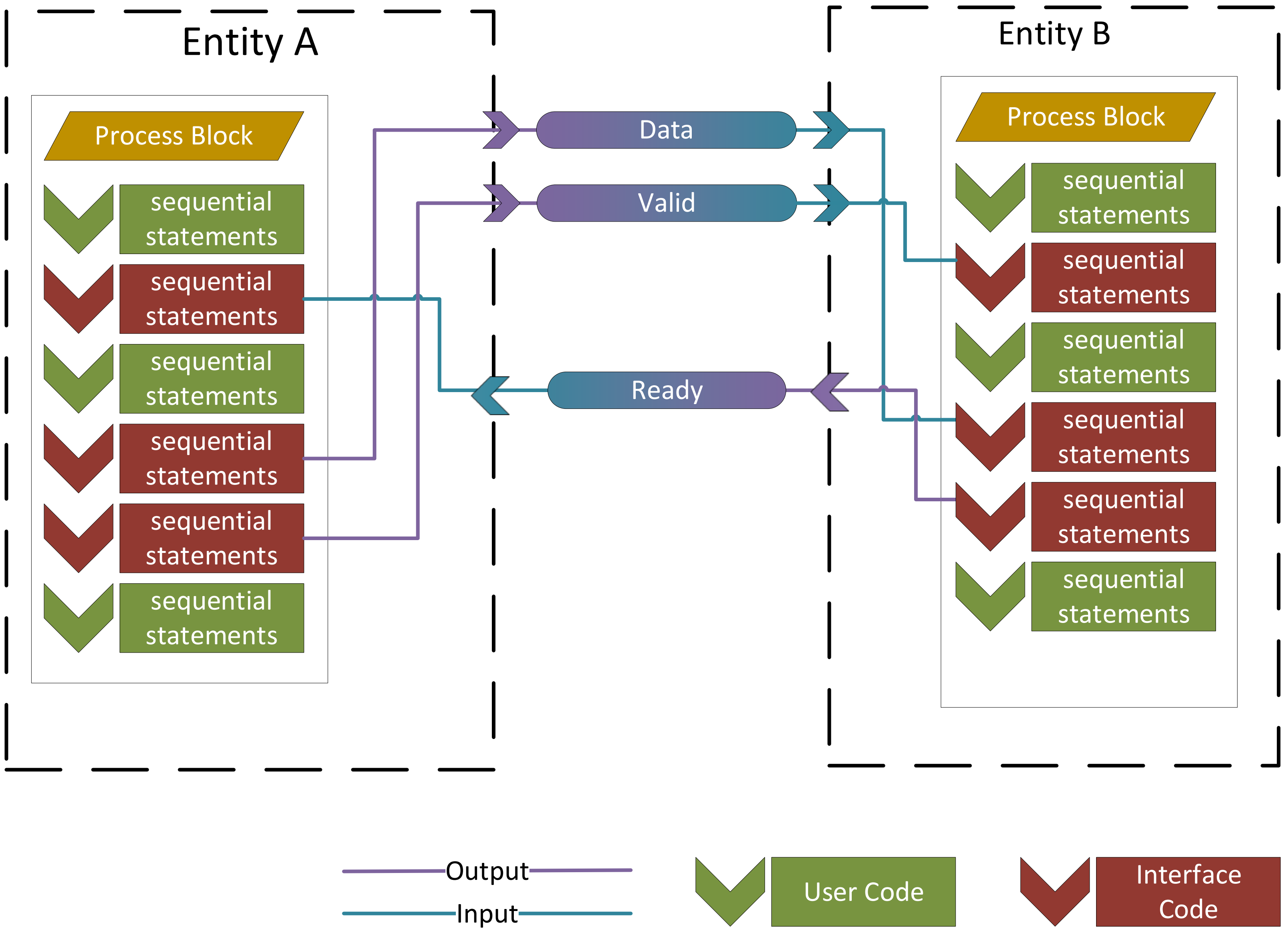
Typical VHDL Entity Declaration

```
entity fifo_cc is
generic(
   DATA_WIDTH : natural := 16;
   DEPTH : natural := 5
);

port(
   clk   : in  std_logic;
   rst   : in  std_logic;
   din   : in  std_logic_vector(DATA_WIDTH-1 downto 0);
   wen   : in  std_logic;
   ren   : in  std_logic;
   dout  : out std_logic_vector(DATA_WIDTH-1 downto 0);
   full  : out std_logic;
   empty : out std_logic
);
end fifo_cc;
```



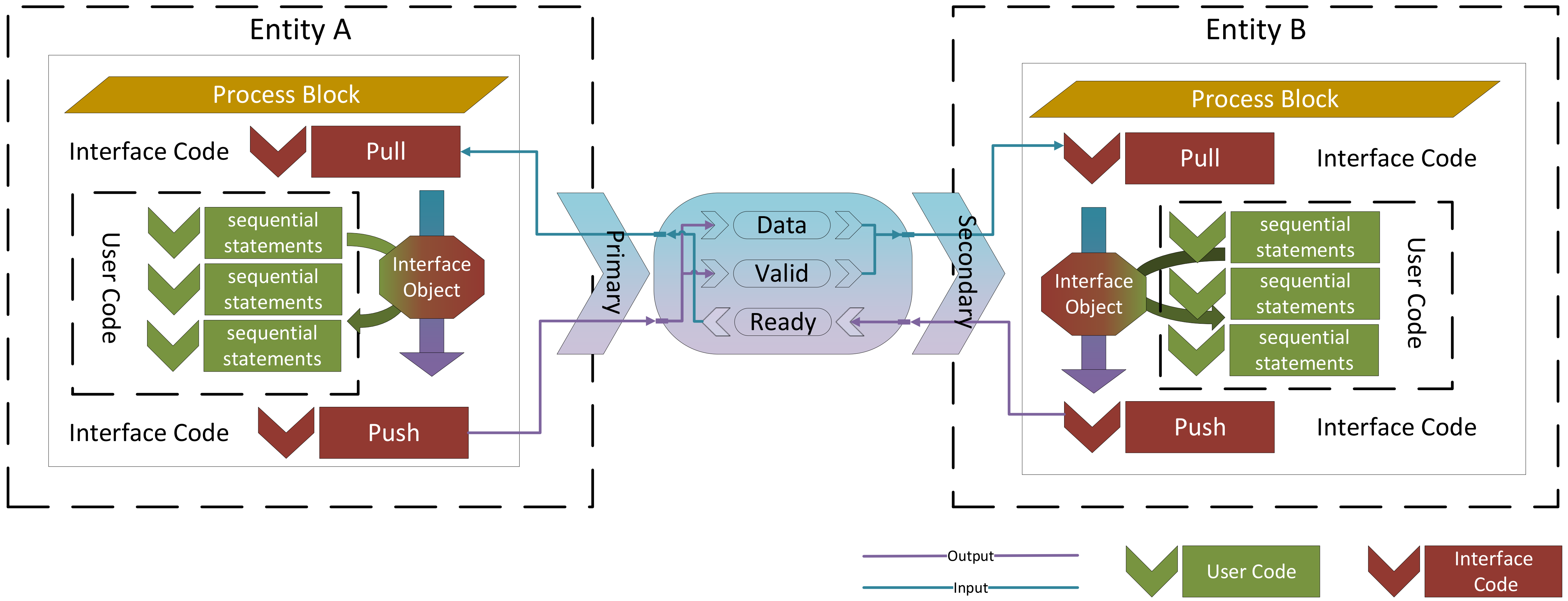
- Consists of Three components:
  - The Entity Name
  - The Generics List (Template Variables)
  - Port List
- Ports Consist of:
  - Name
  - Type
  - Direction (In/Out)

Communication between Entities without Classes



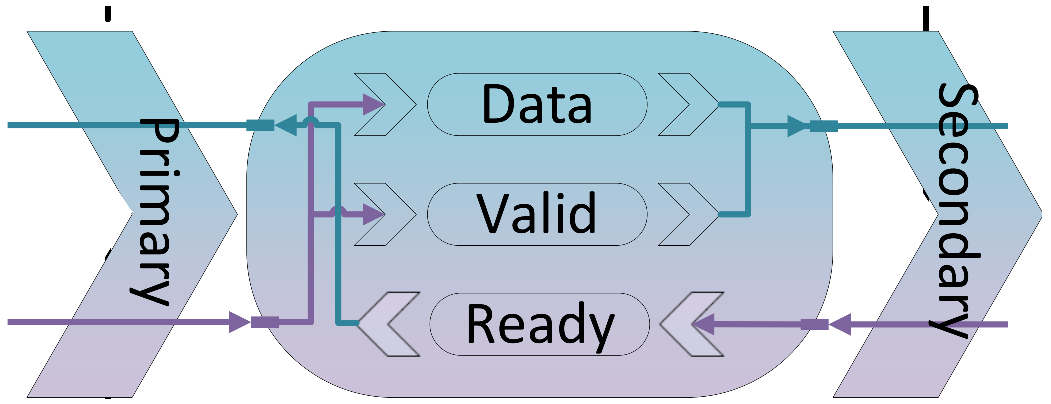
- User Code and Interface code is intermingled
- Library code needs to be re-implemented for each entity
- Library code is hard to recognize
- Inputs and outputs are independent objects

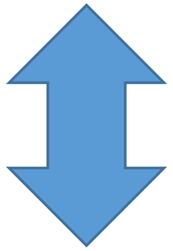
Communication between Entities with Classes



- User Code and Interface code is clearly separated
- Reusing of well known (tested) functions
- Interface code is immediately recognizable

The Interface Class



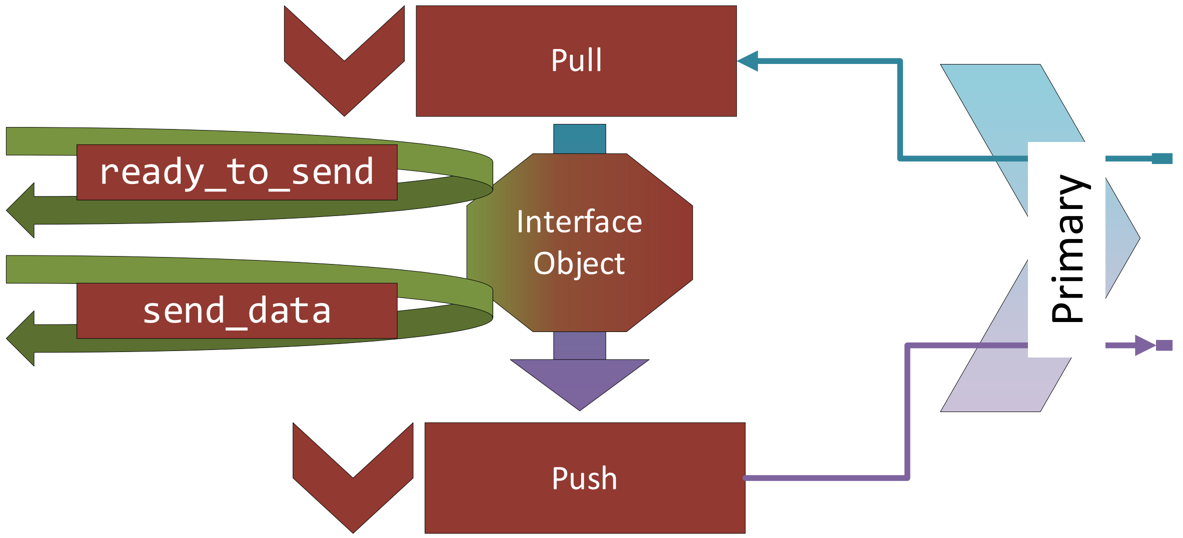


```
class axiStream_32(v_interface):
  def __init__(self):
    super().__init__()
    self.valid = port_out( v_sl()     )
    self.data  = port_out( v_slv(32) )
    self.ready = port_in ( v_sl()     )
```

- This Class describes the signals required for the Interface
- In addition to just storing the type of the data it also stores the direction of the data
- By definition data flows from primary to secondary
  - port_out defines a signal that goes from primary to secondary
  - port_in defines a signal that goes from secondary to primary

Example: Interface Handler Class

```python
class axiStream_sender(v_handler_class):
  def __init__(self, Axi_Out):
    super().__init__()
    self.tx =  v_variable(Axi_Out)
    Axi_Out  << self.tx

  def send_data(self, dataIn ):
    self.tx.valid   << 1
    self.tx.data    << dataIn

  def ready_to_send(self):
    return not self.tx.valid

  def _onPull(self):
    if self.tx.ready:
      self.tx.valid << 0
```

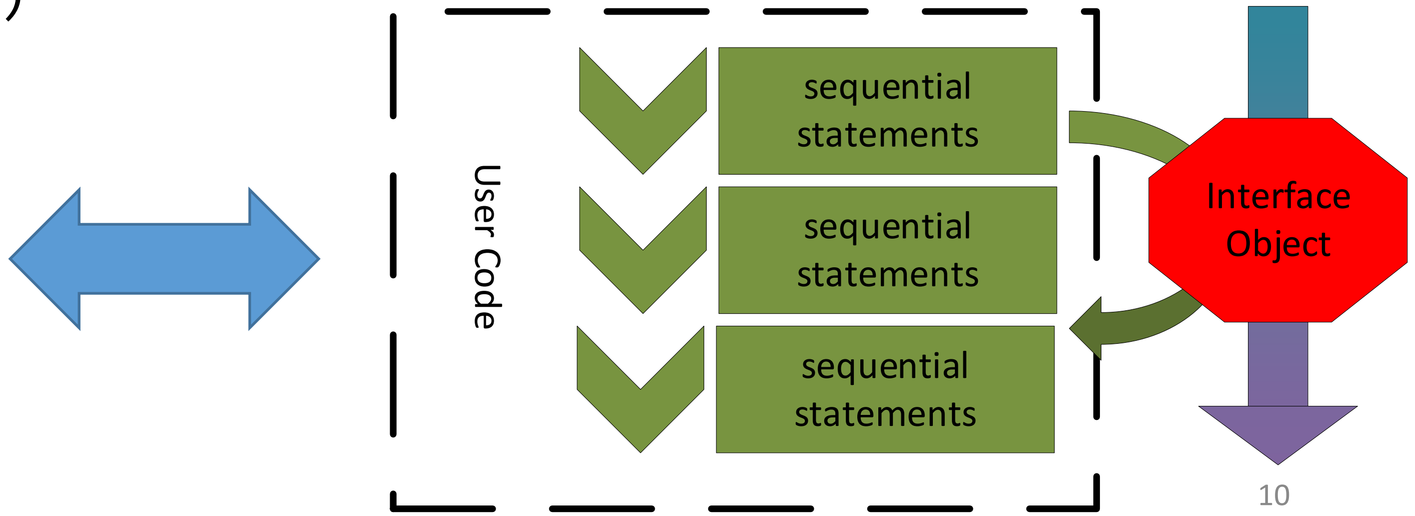


- This Class handles the primary (sender) side of the interface.
- The constructor:
  - takes the Interface class as argument
  - It makes a local variable copy of it
  - It connects the local copy to the Input argument
- The object knows exactly which signals have to go in which direction
- The _onPull functions allows the creator of the library to inject functionality on every clock cycle (before the users code)

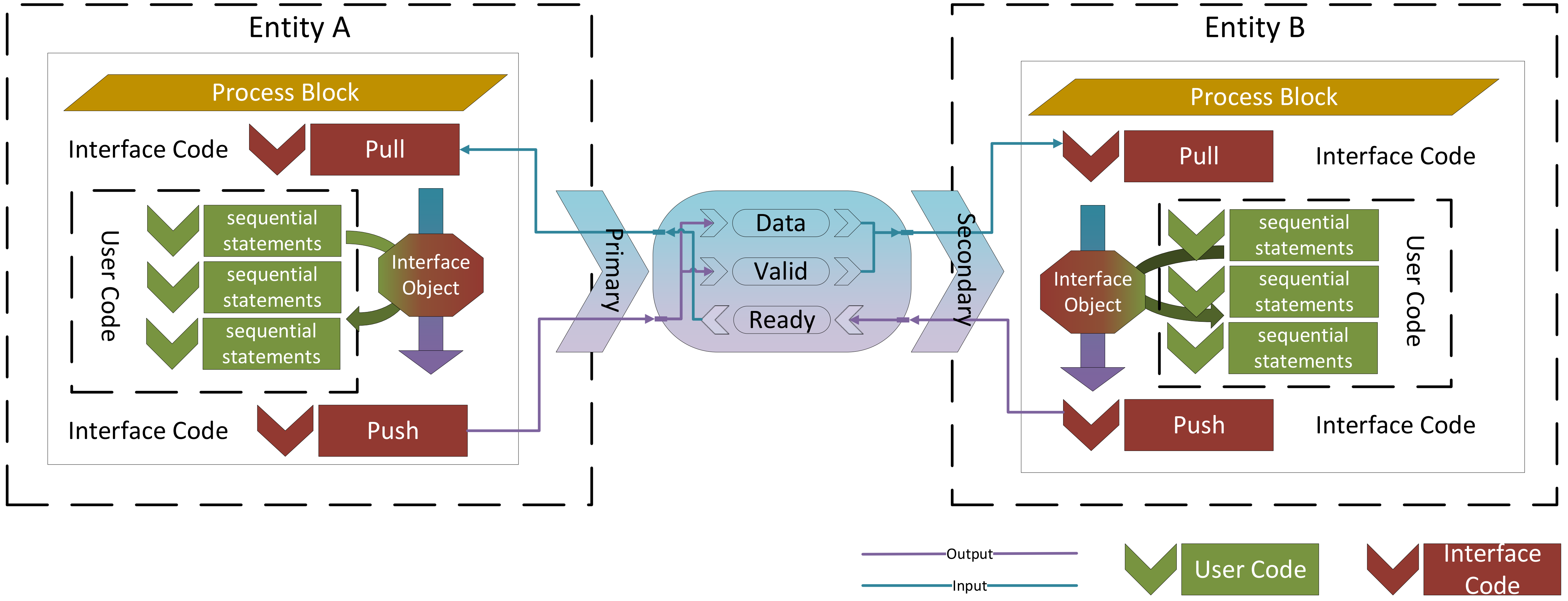
Example: Clocked Entity sending data Via Axi-Stream link

```
from argg_hdl import *
from argg_hdl.examples import *

class Counter(v_clk_entity):
  def __init__(self, clk ):
    super().__init__(clk)
    self.Dout = port_out(axiStream_32())

  @architecture
  def architecture(self):
    data = v_slv(32)
    data_out = get_handle(self.Dout)

    @rising_edge(self.clk)
    def proc():
      if data_out.ready_to_send():
        data_out.send_data(data)
        data << data + 1
```

- Instead of defining individual Inputs and outputs, defining an interface Object
- The Object “axiStream_32” contains all information about input and output signals
- The object brings a handler with it.
- The handler is used for the communication with the interface
- The handler provides the API for the interface
- The user never directly interacts with the data members



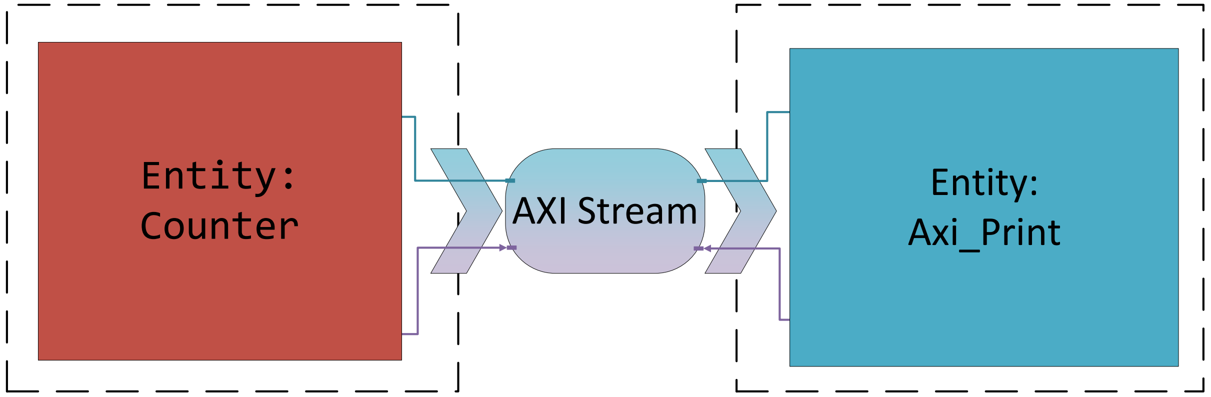
Communication between Entities with Classes



Entities are Objects

```
class tb(v_entity):
  def __init__(self):
    super().__init__()

  @architecture
  def architecture(self):
    clkgen  = clk_generator()
    cnt     = Counter(clkgen.clk)
    axPrint = Axi_Print(clkgen.clk)
    axPrint.D_in << cnt.Dout
    end_architecture()
```

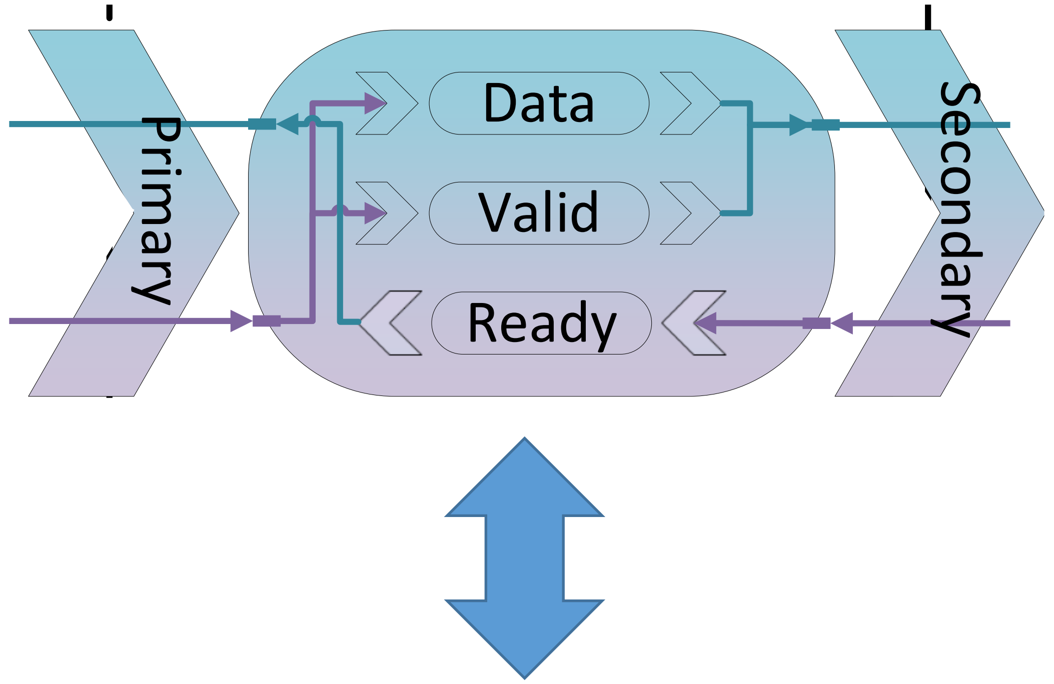
```
class Axi_Print(v_clk_entity):
  def __init__(self, clk ):
    super().__init__(clk)
    self.D_in = port_in(
                   axiStream_32()
                )
```



- Axi_Print:
  - Is an entity that prints out the values it received from the axi stream input
  - It has a slave port of the same interface class as “Counter”
- In order to connect “Counter” with “Axi_Print” the D_in / Dout Member needs to be connected
- Since the interface class knows which signal goes in which direction the signals can just assigned to each other

Generic Code through Templates and customizable “HDL Converters“

Generic Interfaces Classes



```
class axiStream(v_interface):
  def __init__(self, Axitype):
    super().__init__()
    self.valid  = port_out( v_sl()   )
    self.data   = port_out( Axitype )
    self.ready  = port_in ( v_sl()   )
```

- ARGG HDL:
  - preserves as much as possible the generic programming model of python
  - All class members are generic objects
  - The actual type of an object and its member is only defined once it is created
  - All function are generic / template function
- Example: Interface Classes
  - For the actual implementation of the Axi Stream interface class the data object is a template
  - For each new Data Type the templating machine will create a new class
  - Data type can also be records and arrays

Example: Interface Handler Classes

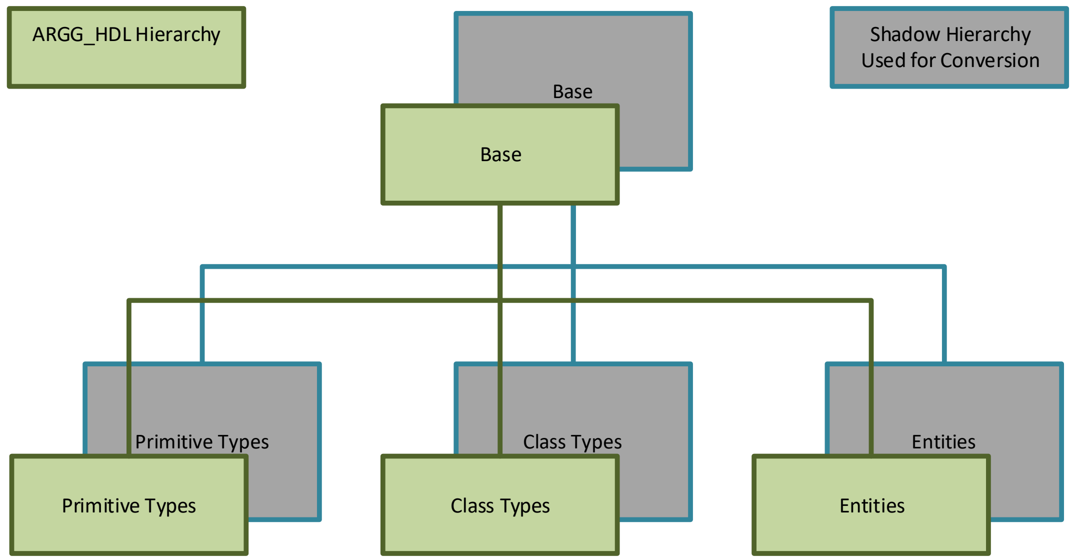
```
class tb(v_entity):
    @architecture
    def architecture(self):
        clkgen   = clk_generator()
        cnt      = Counter(clkgen.clk)
        cnt_out  = get_handle(cnt.Data_out)
        data     = v_slv(32)
        opt_data = optional_t(v_slv(32))

        @rising_edge(clkgen.clk)
        def proc():
            cnt_out >> data
            cnt_out >> opt_data

        end_architecture()
```

```
class axisStream_receiver(v_handle_class):
  def __rshift__(self, rhs):
    rhs.reset()
    if self.data_internal_isvalid:
      rhs << self.data_internal
      self.data_internal_was_read << 1
```

- Optional_t is a generic container that stores data of a certain type as well as a bit that indicates if the data is valid or not
- cnt_out is an instance of axisStream_receiver. It was written long before optional_t.
- Since optional_t has a reset function and an assignment operator that accepts this data type it will compile just fine

Generic Conversion to VHDL

ARGG HDL Object Hierarchy



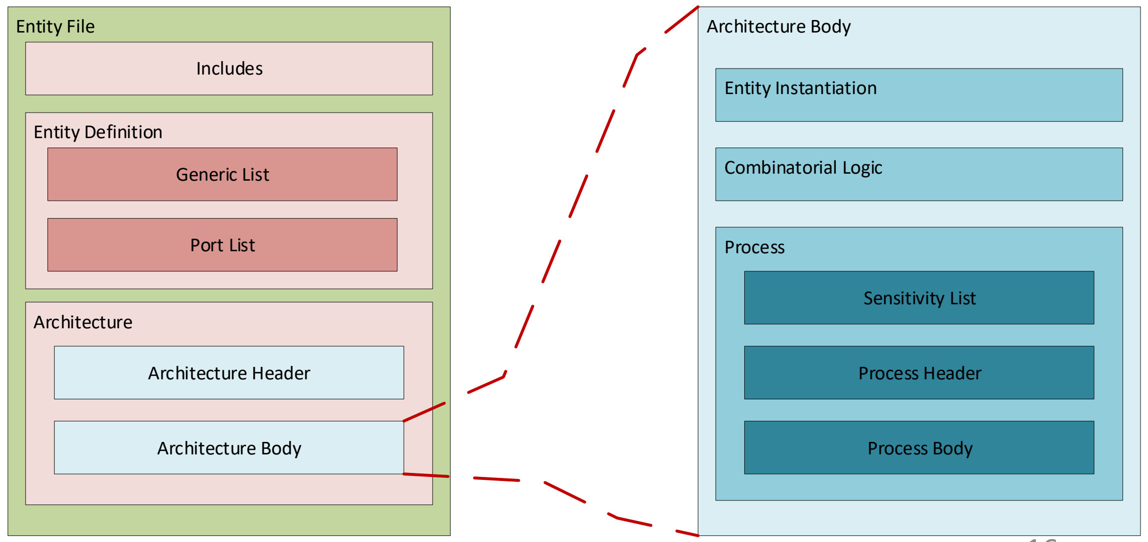
HDL Converter Object

- For the conversion to VHDL each object has a **HDL Converter Object**
- **HDL Converter Object** have the same inheritance hierarchy as the object it belongs to. (Shadow Hierarchy)
- Only Code inside the **HDL Converter Object** is language specific

Code Injection

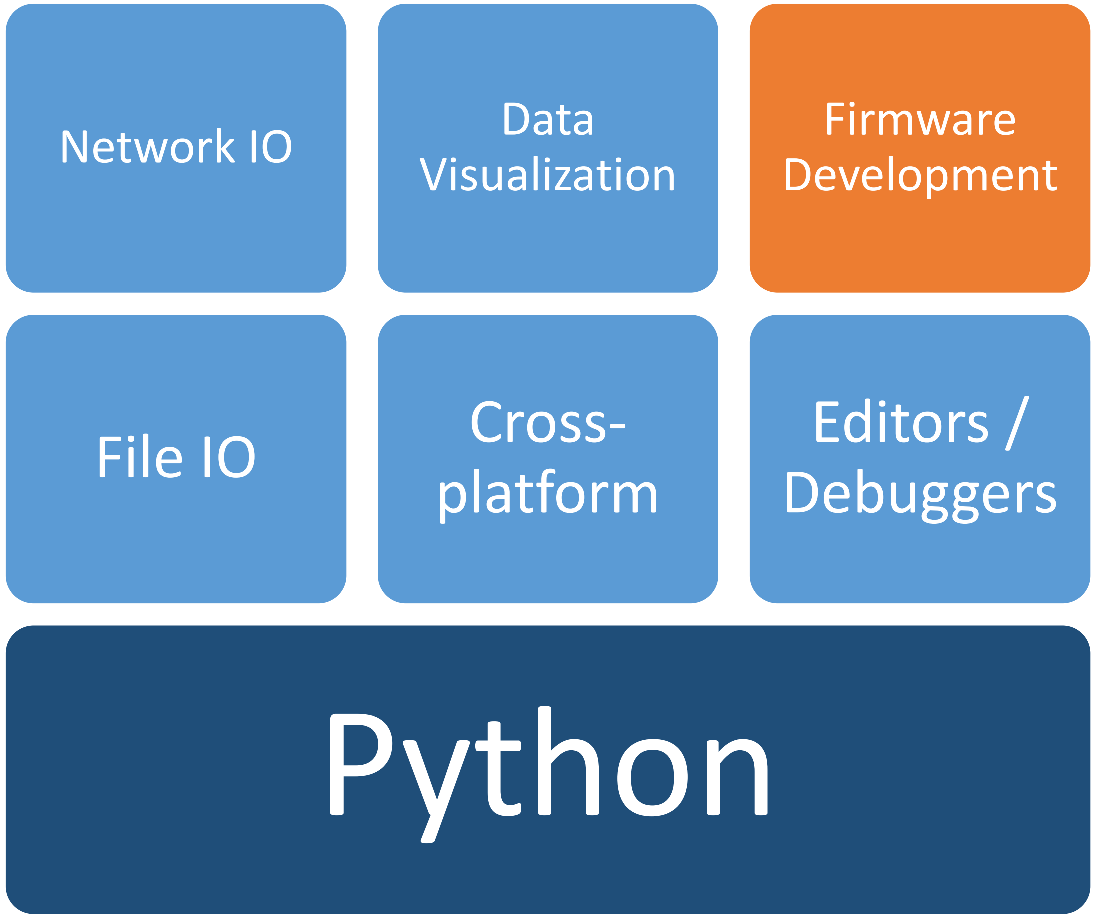
- **Entities** in VHDL consist of the blocks shown on the left
- The **HDL Converter Object** of each **ARGG_HDL** object has a function overload that can inject code at any of these location
- The **Python** code and the **VHDL** code can be made completely **independent** of each other

Code Injection



Powerful Tools by using an existing Language

ARGG_HDL is Powered by



- ARGG_HDL is not a language; it is a library for a language

➔ Simulation runs **Cross-Platform** as Python script

➔All tools and libraries for python can be used for ARGG_HDL
- ➔ **Step Debugging**
- ➔ **Data Visualization** with Matplotlib, Plotly
- ➔ **File IO** with pyvcd, pandas or ROOT

➔Python has interface to virtually all other popular languages
- ➔**Co-simulation**: either through language interface or through TCP/IP sockets (or …)

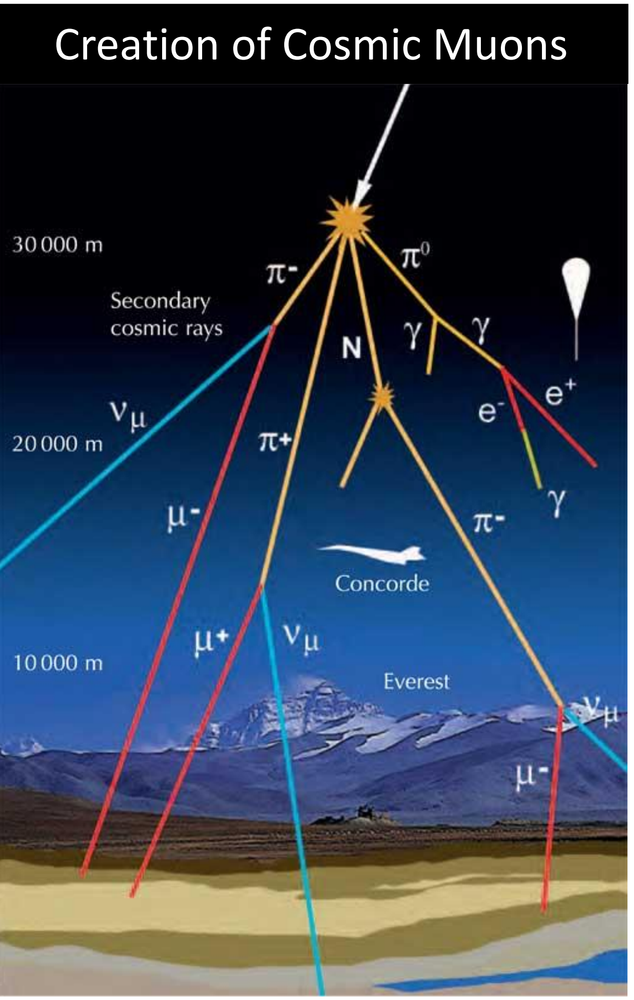
➔Firmware development using a **familiar language**

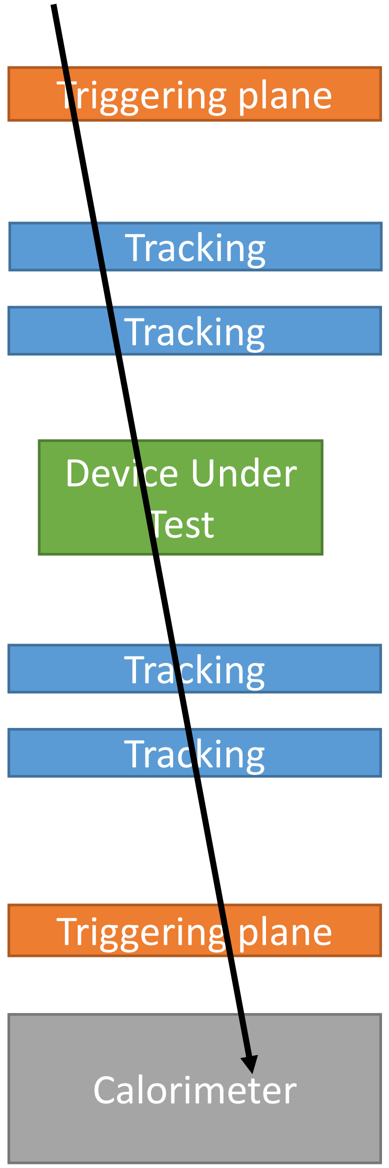
➔Python has a **packaging system**
- ➔Convenient code sharing

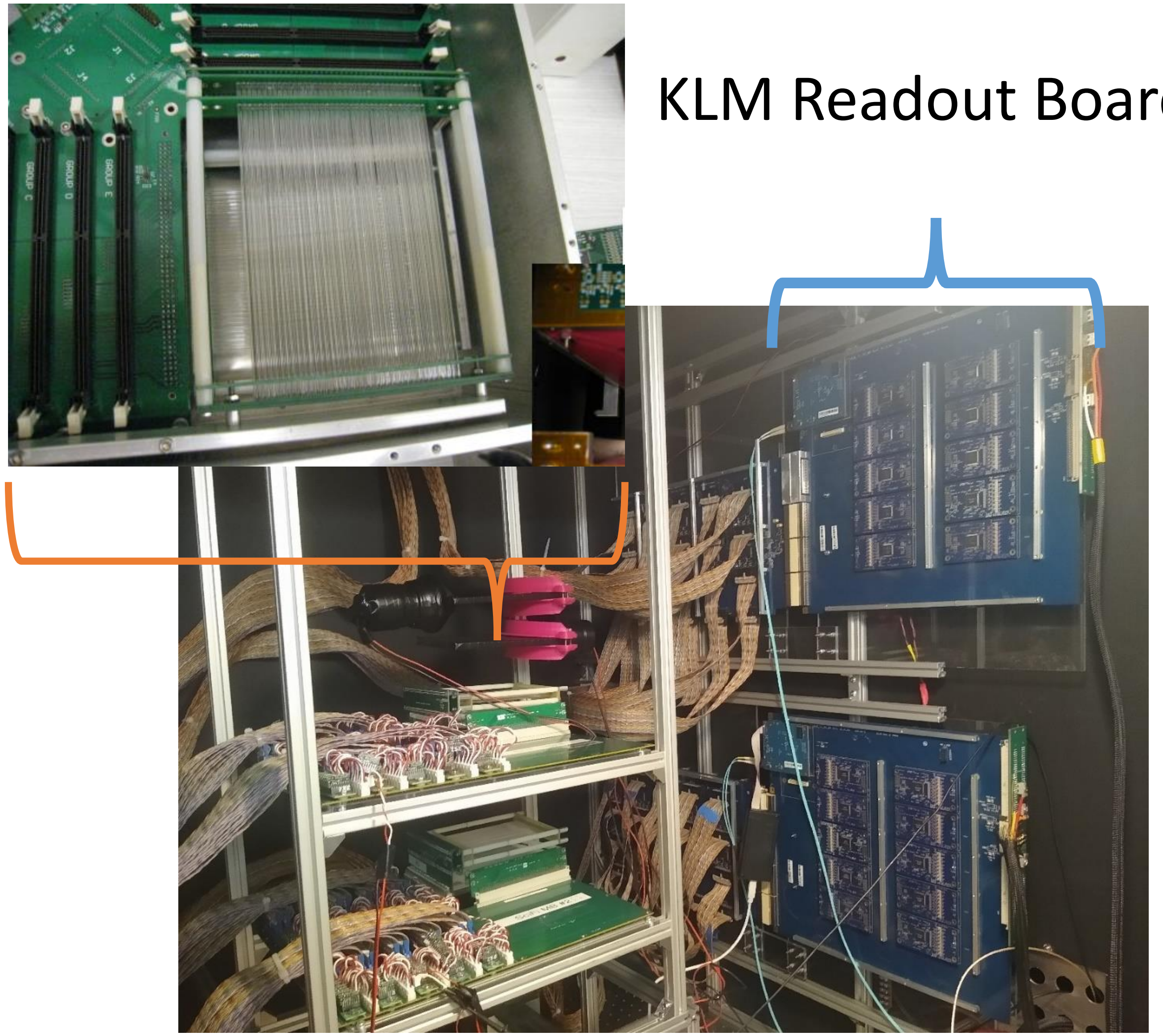
End

Backup

Hawaiian Muon Beamline (HMB) v3



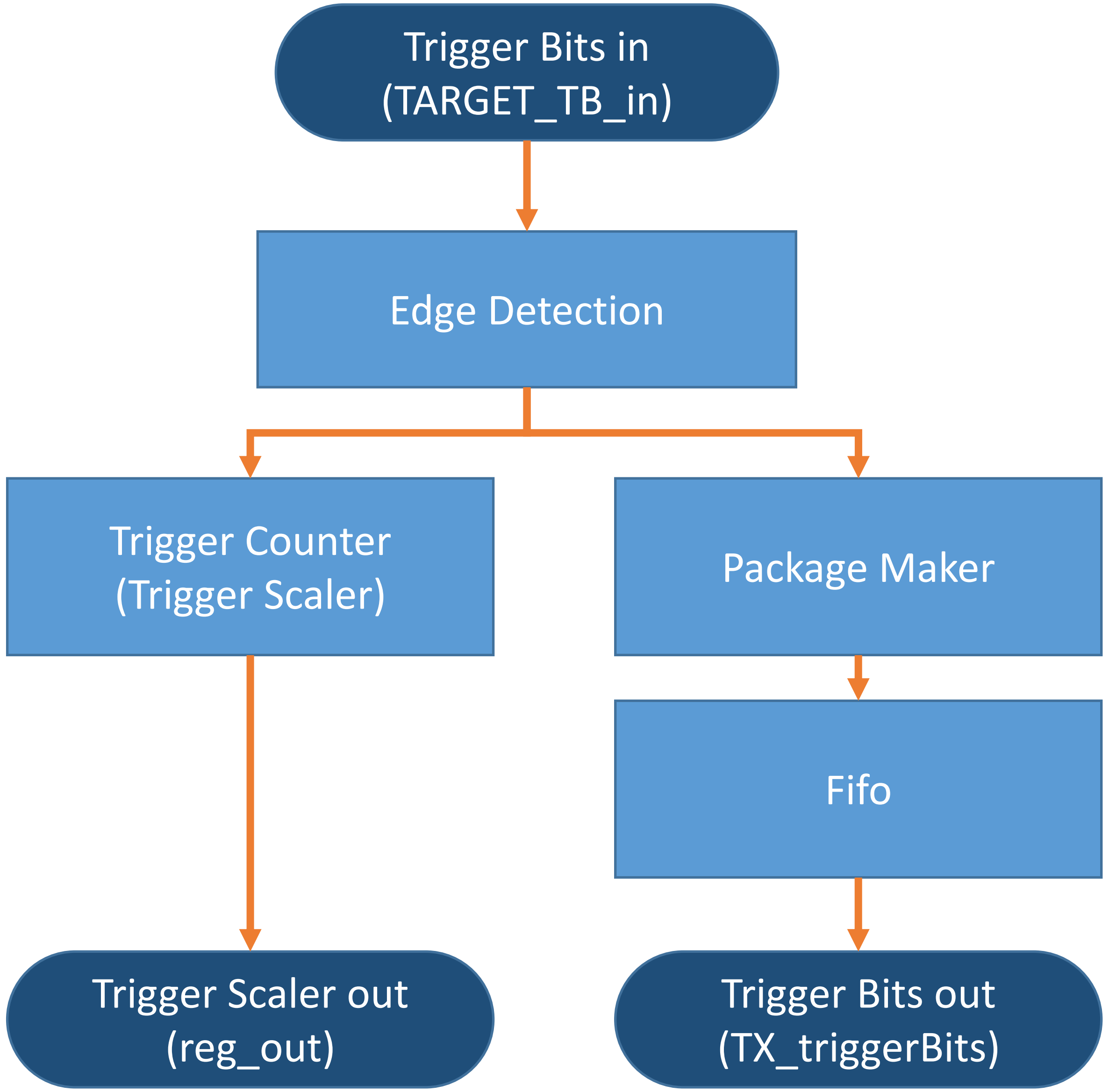




Richard Peschke | rp40@Hawaii.edu

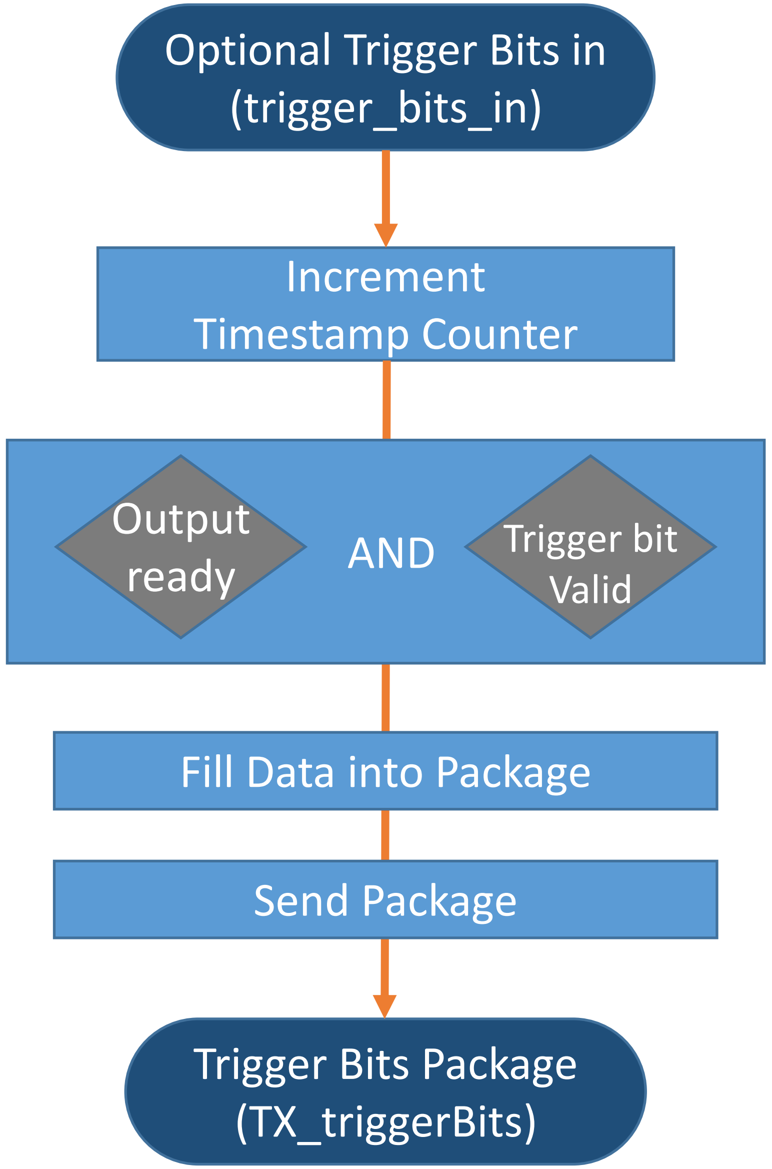
Hawaiian Muon Beamline (HMB) and KLM Readout Boards

```python
class TX_TriggerBitSZ(v_entity):
    def __init__(self, gSystem:globals_t):
        self.gSystem = port_in(gSystem)
        self.gSystem << gSystem
        self.reg_out  = port_out(registerT())
        self.TARGET_TB_in  = port_in(tb_vec_type())
        self.TX_triggerBits = port_out(axi_Stream(trigger_bits_pack()))

        self.architecture()

    @architecture
    def architecture(self):
        edge_det = tb_edge_detection(self.gSystem)

        self.TARGET_TB_in \
            | \
        edge_det

        edge_det \
            | trigger_scaler(self.gSystem)  \
            |\
        self.reg_out

        edge_det\
            | package_maker(self.gSystem )  \
            | ax_fifo(self.gSystem.clk, trigger_bits_pack())\
            | \
        self.TX_triggerBits

        end_architecture()
```



Hawaiian Muon Beamline (HMB) and KLM Readout Boards

```python
class optional_trigger_bits(v_record):
    def __init__(self) -> None:
        self.TriggerBits =  tb_vec_type()
        self.valid       =  v_sl()



class package_maker(v_entity):
    def __init__(self, gSystem:globals_t, reg_out :registerT) -> None:
        self.gSystem = port_in(gSystem)
        self.gSystem << gSystem

        self.trigger_bits_in  = pipeline_in(optional_trigger_bits())
        self.TX_triggerBits   = pipeline_out(
                                    axi_Stream(trigger_bits_pack())
                                    )

        self.architecture()

    @architecture
    def architecture(self):
        counter = v_slv(64)
        tx = get_handle(self.TX_triggerBits)
        buff = v_variable(trigger_bits_pack())
        @rising_edge(self.gSystem.clk)
        def proc():
            counter << counter + 1
            if tx and self.trigger_bits_in.valid:
                buff.time_stamp<< counter[32:]
                buff.time_stamp_fine<< counter[0:32]
                buff.data << self.trigger_bits_in.TriggerBits

                tx << buff
```


Optional Trigger Bits in (trigger_bits_in)
Increment Timestamp Counter
Output ready
AND
Trigger bit Valid
Fill Data into Package
Send Package
Trigger Bits Package (TX_triggerBits)

End