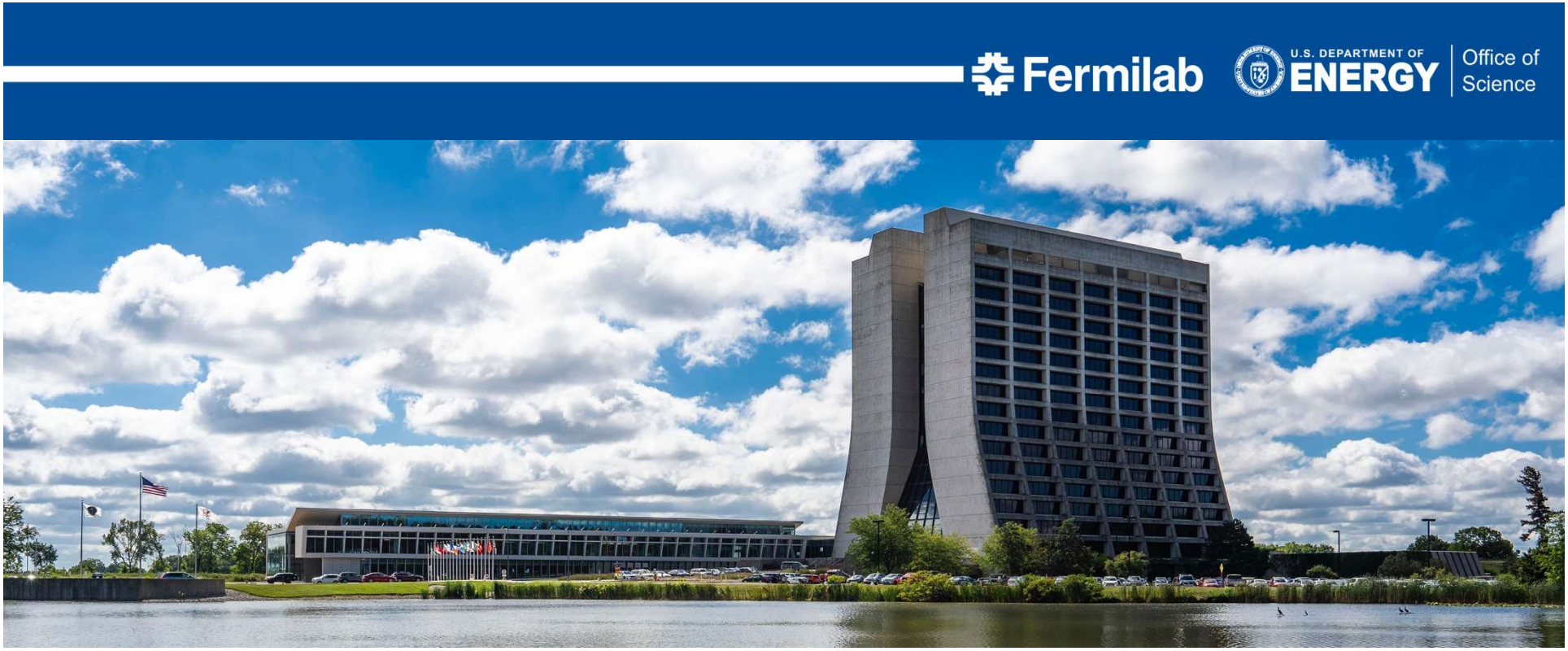



# Enhancing ASIC Verification for HEP with Cocotb and PyUVM

Manish Rangarajan Shankar
Columbia University
Micro-Electronics Division, Fermilab, USA
9th July 2024

# Outline


1. Current Challenges of ASIC verification in HEP
2. What is Cocotb?
3. Advantages over UVM
4. Fast Command Controller Specification
5. Coroutine based testbench design
6. PyUVM based testbench design
7. Conclusion

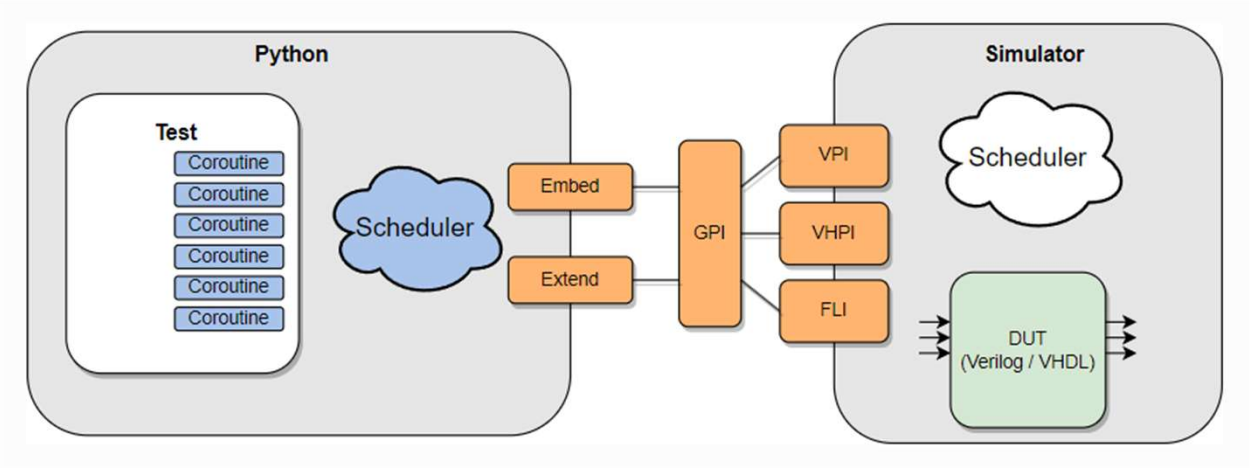
# Current challenges for ASIC Verification at HEP

- ASIC Development for HEP application often face constrains on engineering costs for ASIC design and verification
- The current ratio of engineering costs for ASIC design to verification is almost 1:3
- Verification Methodology is the bottleneck
- Over the last few years, IP/ASIC/SoCs functional verification has been conducted using Universal Verification Methodology (UVM)
- UVM is standardized methodology based upon System Verilog (SV) which has been traditionally used for RTL design
- **Not many in HEP field know System Verilog/UVM !**
- This makes it complex ASIC verification a resource constrained task!

References:

1. UVM (Universal Verification Methodology) (acclelera.org)

# What is Cocotb? How does it work?



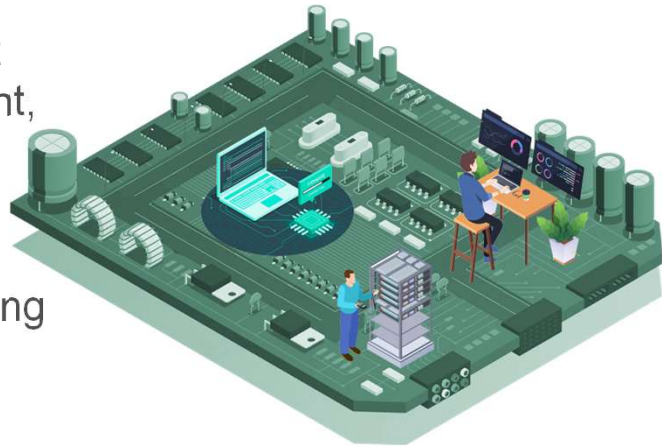

1. Cocotb [1] interacts with HDL similators (Icarus, Xcelium, Modelsim, etc.) using either FLI (Foreign Language Interface) or VPI (Verilog Procedural Interface) or VHPI ( Verilog Hardware Procedural Interface)
2. Cocotb uses an even-driven simulation model
3. It means that simulation progresses based on events such as RisingEdge/FallingEdge of RTL signals or coroutine scheduling rather than advancing in lockstep with a clocks.
4. Briefly coroutines are concurrent processes in testbench and are scheduled by Cocotb's internal scheduler
5. This scheduler is responsible for determining the order of execution of the coroutines based upon events and inter-dependencies.

References:

1. Quickstart Guide --- cocotb 1.8.1 documentation

## Why is it necessary? Advantage over current methodology

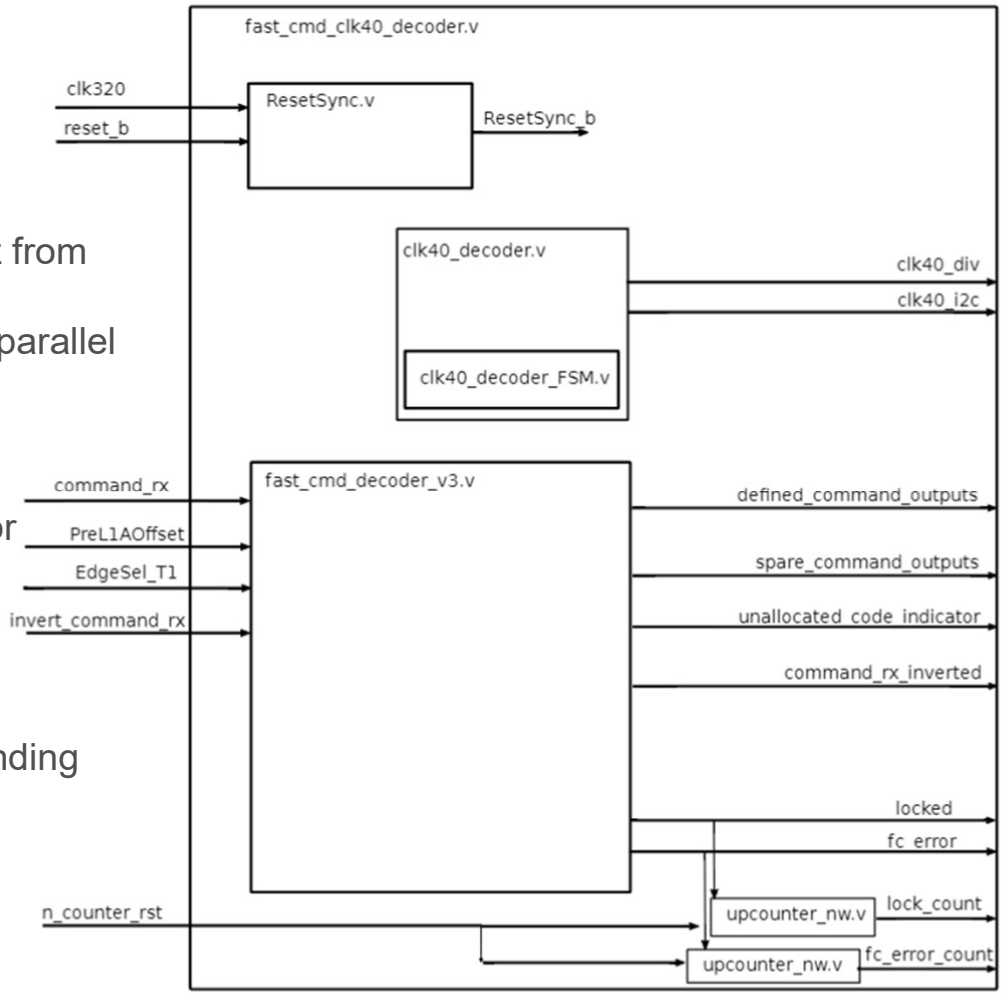
- **Python Integration:** Seamlessly integrates with Python, allowing for rapid testbench development and leveraging Python's extensive ecosystem.
- **Coroutines for Simplicity:** Utilizes coroutines for testbench processes, simplifying asynchronous and event-driven simulation compared to traditional procedural approaches.
- **Open-Source Community Support:** Backed by a vibrant open-source community, ensuring continuous improvement, bug fixes, and support for various simulators.
- **Enhanced Productivity:** Reduces development time through automated testing and easy integration with existing design flows, improving overall verification efficiency.
- **Flexible and Scalable:** Adaptable to various design sizes and complexities, promoting scalable verification environments without compromising on flexibility.



# Fast Command Controller Specification - I

## Description

- Peripheral IP Functionality
  - Samples 8-bit serial commands at 320MHz from an external IP (e.g., lpGBT).
  - Decodes these serial commands into 8-bit parallel commands at 40MHz.
- High-Speed Operation
  - Operates at a high frequency of 320MHz for sampling input commands.
- Command Handling
  - Handles 42 distinct input commands.
  - Decodes and outputs through 27 corresponding output pins.




RTL Block Diagram

CERN CMS FC Controller Documentation

# Fast Command Controller Specification - II

Valid Command Codes

| command | code |
|---|---|
| CMD_IDLE | 00110110 |
| CMD_not_IDLE | 11001001 |
| CMD_PREL1A | 11001100 |
| CMD_L1A | 01001011 |
| CMD_L1A_PREL1A | 11100001 |
| CMD_L1A_NZS | 00001111 |
| CMD_L1A_NZS_PREL1A | 00101011 |
| CMD_L1A_BCR | 01110001 |
| CMD_L1A_BCR_PREL1A | 10100101 |
| CMD_L1A_CALPULSEINT | 00111001 |
| CMD_L1A_CALPULSEEXT | 10000111 |
| CMD_L1A_CALPULSEINT_PREL1A | 11100010 |
| CMD_L1A_CALPULSEEXT_PREL1A | 11110000 |
| CMD_BCR | 00011101 |
| CMD_BCR_PREL1A | 10100011 |
| CMD_BCR_OCR | 10010101 |
| CMD_CALPULSEINT | 00101101 |
| CMD_CALPULSEEXT | 01111000 |
| CMD_CALPULSEINT_PREL1A | 01010101 |
| CMD_CALPULSEEXT_PREL1A | 10010011 |
| CMD_CHIPSYNC | 11010010 |
| CMD_EBR | 11010001 |
| CMD_ECR | 10101001 |
| CMD_LINKRESETROCT | 10011001 |
| CMD_LINKRESETROCD | 10011010 |
| CMD_LINKRESETECONT | 10101010 |
| CMD_LINKRESETECOND | 10111000 |

FC Commands

CERN CMS FC Controller Documentation

# COCOTB for FC Controller

## Makefile & File Structure

- File Structure Overview
  - Key directories: constants, doc, filelists, scripts, subIP, vrf
  - Important files: Project_constants.sv, README.md, various documentation and testbench files
- Makefile Configuration
  - Sets PYTHONPATH for module paths
  - Defines directories and includes for filelists and sources
  - Uses cocotb-config and tclsh for configuration and parsing
- Clean and Build Targets
  - clean_all: Removes build artifacts and __pycache__
  - tbtop: Cleans all and runs the simulation

```
export PYTHONPATH := $(REPO_ROOT)/vrf/tb/$(MODULE):$(REPO_ROOT)/vrf/uvcs/$(MODULE)_uvc:$(PYTHONPATH)
SIM_DIR                         = $(REPO_ROOT)/vrf/cocotb
RUN_DIR                         = $(SIM_DIR)/run
SCRIPT_DIR                    = $(REPO_ROOT)/scripts
FILELISTS_DIR               = $(REPO_ROOT)/filelists
SRC_TBTOP             = $(FILELISTS_DIR)/sim_tbtop.f
SIM_BUILD                       = $(RUN_DIR)/$(MODULE)
VERILOG_INCLUDE_DIRS := $(REPO_ROOT)/subIP/fast_control_v3/src/ $(REPO_ROOT)/constants/ $(REPO_ROOT)/vrf/tbtop/
```

```
.SILENT: clean_all
clean_all: clean
    $(RM) -r $(REPO_ROOT)/vrf/tb/$(MODULE)/__pycache__
    $(RM) -r $(TOPLEVEL).xml

.SILENT: tbtop
tbtop:
    $(MAKE) clean_all && $(MAKE) sim
```

```
[fc_controller_cocotb]
├── CHANGELOG.md
├── constants
│   └── Project_constants.sv
├── doc
│   ├── fc_controller.docx
│   ├── FC_Controller.drawio
│   ├── fc_controller.pdf
│   ├── FCC_TB.drawio
│   └── README.md
├── filelists
│   ├── rtl.f
│   ├── sim_tbtop.f
│   └── tbtop.f
├── README.md
├── scripts
│   └── parse_filelist.tcl
├── sim_var_local.mk
├── subIP
│   └── fast_control_v3
└── vrf
```

```
└── vrf
    ├── cocotb
    │   └── run
    │       └── Makefile
    ├── tb
    │   └── fc_controller_tb
    │       └── fc_controller_tb.py
    ├── tbtop
    │   └── top_test_harness.sv
    └── uvcs
        └── fc_controller_tb_uvc
            ├── fc_coroutines.py
            ├── fc_coverage.py
            ├── fc_driver.py
            ├── fc_env.py
            ├── fc_mon.py
            ├── fc_scoreboard.py
            └── __init__.py
```

References:
1. Makefile from Bootstrap cocotb-repo



Fermilab

# COCOTB for FC Controller

## Coroutines and Examples

- What is a Coroutine?
  - A coroutine is a special type of function that can pause and resume its execution.
  - Enables asynchronous operations, useful for event-driven programming.
  - In cocotb, coroutines are used to create structured, non-blocking testbench code.

- ClockGen
  - Generates a clock signal based on user-specified or default frequency (320MHz).
  - Logs clock details and starts the clock signal.

- ResetDut
  - Performs synchronous reset of the DUT, supporting both hard and soft reset modes.
  - Logs reset type and status, asserts and de-asserts reset signals.

```python
@cocotb.coroutine
def ClockGen(dut):
  cocotb.log.info(f"--------------------------------------------------------------------------------")
  cocotb.log.info(f"------------------------ CLOCK-----------------------------------------------")
  cocotb.log.info(f"--------------------------------------------------------------------------------\n")

  clk = cocotb.plusargs.get("CLK")
  if clk is not None:
    clk = int(clk)
    clk_period = int(1e9/(clk*1e6))
    cocotb.log.info(f"Using clk specified by user: {clk} MHz with period {clk_period} ns \n")
  else:
    clk = 320
    clk_period = int(1e9/(clk*1e6))
    cocotb.log.info(f"No clk frequency specified by user, using {clk} MHz with period {clk_period} ns \n")

  cocotb.start_soon(Clock(dut.clk320_tb, clk_period, units="ns").start())
  yield RisingEdge(dut.clk320_tb)
```

```python
@cocotb.coroutine
def ResetDut(dut):
  cocotb.log.info(f"--------------------------------------------------------------------------------")
  cocotb.log.info(f"------------------------ RESET-----------------------------------------------")
  cocotb.log.info(f"--------------------------------------------------------------------------------\n")

  yield RisingEdge(dut.clk320_tb)
  reset = cocotb.plusargs.get("RESET","HARD")
  cocotb.log.info(f"{reset} reset is requested, reset is synchronous active LOW \n")
  if reset == "HARD":
    dut.n_rstExt_tb.value = 0
    dut.n_rstExtSoft_tb.value = 0
    yield RisingEdge(dut.clk320_tb)
    dut.n_rstExt_tb.value = 1
    dut.n_rstExtSoft_tb.value = 1
    cocotb.log.info(f" De-asserted both n_rstExt and n_rstExtSoft as HARD reset is requested \n")
    for _ in range(10):
      yield RisingEdge(dut.clk320_tb)
  elif reset == "SOFT":
    dut.n_rstExtSoft_tb.value = 0
    dut.n_rstExt_tb.value = 0
    yield RisingEdge(dut.clk320_tb)
    dut.n_rstExtSoft_tb.value = 1
    cocotb.log.info(f" De-asserted only n_rstExtSoft as SOFT reset is requested \n")
```

# COCOTB for FC Controller

## Key Coroutines

```python
@cocotb.coroutine
def SigInitialize(dut):
    cocotb.log.info(f"--------------------------------------------------------------------------------")
    cocotb.log.info(f"------------------------ SIGNAL INITIALIZATION-----------------------------")
    cocotb.log.info(f"--------------------------------------------------------------------------------\n")

    yield RisingEdge(dut.clk320_tb)
    # Inputs
    dut.n_counter_rst_tb.setimmediatevalue(0)
    dut.EdgeSel_T1_tb.setimmediatevalue(0)
    dut.PreL1AOffset_tb.setimmediatevalue(0)
    dut.invert_command_rx_tb.setimmediatevalue(0)
```

- SigInitialize
  - Initializes input and output signals to their default states after a reset.
  - Ensures the DUT starts with known, stable signal values.

- SendFC_Idle
  - Sends the IDLE command five times to initialize the DUT properly.

- SendFC
  - Sends a specific fast command serially via the command_rx line.
  - Converts command data to bits and transmits them at the clock edge.

- SendRandomFC
  - Chooses and sends a random fast command from a predefined list.
  - Useful for stress testing and validating DUT response to various commands.

```python
@cocotb.coroutine
def SendFC_Idle(dut):
    data_in = 0x36
    cocotb.log.info(f" Sending IDLE command serially through command_rx \n")
    for _ in range(5):
        for i in range(7, -1, -1):
            bit_to_send = (data_in >> i) & 1
            dut.command_rx_tb.value = bit_to_send
            yield RisingEdge(dut.clk320_tb)
        yield RisingEdge(dut.clk320_tb)
    for i in range(7):
        yield RisingEdge(dut.clk320_tb)


@cocotb.coroutine
def SendFC(dut, cmd, data):
    data_in = data
    for i in range(7, -1, -1):
        bit_to_send = (data_in >> i) & 1
        dut.command_rx_tb.value = bit_to_send
        yield RisingEdge(dut.clk320_tb)
    dut.command_rx_tb.value = 0
    yield RisingEdge(dut.clk320_tb)

    for i in range(39):
        yield RisingEdge(dut.clk320_tb)
```

# COCOTB for FC Controller

## Cocotb Test Implementation

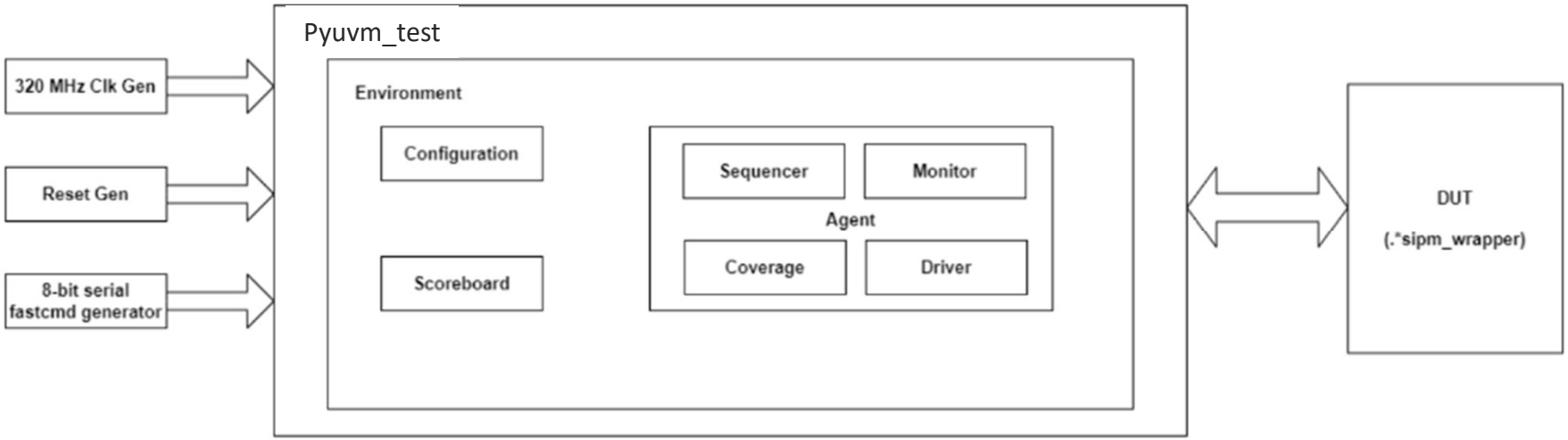
- Cocotb Test Decorator: @cocotb.test()
  - Purpose: Marks a function as a test that will be executed by the cocotb test framework.
  - Usage: Applied to functions that define the test logic for the DUT.
  - Benefits: Simplifies test organization and execution, integrates seamlessly with cocotb's coroutine-based structure.
- Test Function Structure
  - Logging: Provides information about the test being run and the DUT.
  - Coroutine Calls: Sequentially calls coroutines.

```python
@cocotb.test()
def RandomFCTest(dut):
    cocotb.log.info(f"DUT name: {dut._name}")
    cocotb.log.info(f"-----------------------------------
    cocotb.log.info(f"---Running Test: RandomFCTest ---")
    cocotb.log.info(f"-----------------------------------

    yield ClockGen(dut)
    yield ResetDut(dut)
    yield SigInitialize(dut)
    yield SendFC_Idle(dut)
    yield SendRandomFC(dut)
    yield RisingEdge(dut.clk320_tb)

factory = TestFactory(RandomFCTest)
factory.generate_tests()
```

# PYUVM for FC Controller




Proposed PyUVM Testbench

# PYUVM for FC Controller

## FC_Test and FC_Env

- PyUVM Test Implementation
  - Defines a UVM test class using the @pyuvm.test() decorator.
  - Implements the build phase to set up the test environment.
  - Implements the run phase to execute the test sequence.
- FC Env Build Phase
  - Initializes the UVC Components
    - DUT instance retrieval.
    - Configures sequencer communication.
- Connect Phase
  - Links UVC components using analysis ports
- Run Phase
  - Controls simulation:
    - Stops driver.

```python
@pyuvm.test()
class FCTest(uvm_test):
    def build_phase(self):
        self.env = FCEnv("env", self)

    async def run_phase(self):
        self.raise_objection()
        test_seq = TestAllSeq("test_seq")
        await test_seq.start(self.env.seqr)
        self.drop_objection()
```

```python
class FCEnv(uvm_env):
    def build_phase(self):
        self.dut = cocotb.top
        self.cmd_mon = FCMonitor("cmd_mon", self, self.dut)
        self.seqr = uvm_sequencer("seqr", self)
        ConfigDB().set(None, "*", "SEQR", self.seqr)
        self.driver = FCDriver.create("driver",self)
        self.driver.dut = self.dut
        self.scoreboard = FCScoreboard("scoreboard", self, self.dut)
        self.coverage = FCCoverage("coverage", self)


    def connect_phase(self):
        self.driver.seq_item_port.connect(self.seqr.seq_item_export)
        self.cmd_mon.ap.connect(self.scoreboard.result_export)
        self.driver.ap.connect(self.scoreboard.cmd_export)
        self.driver.ap.connect(self.coverage.cmd_export)

    async def run_phase(self):
        for i in range(100):
            await RisingEdge(self.dut.clk320_tb)
        self.driver.stop()
```

# PYUVM for FC Controller

## FCDriver (PYUVM Driver)

- Purpose
  - Drives stimulus to the DUT and manages sequences to validate functionality.
- Initialization
  - Sets up the analysis port (ap) to communicate with other components.
- Run Phase
  - Executes sequences:initialize_sequence(): Initializes the DUT with necessary signals.randomfc_sequence(): Sends random fast commands to the DUT.
  - Manages sequence flow and handles different command transactions (Initialize, RandomFC).
- Stop Mechanism
  - Signals when to stop processing sequences to ensure controlled test execution.

```python
class FCDriver(uvm_driver):
    def __init__(self, name, parent):
        super().__init__(name, parent)
        self.dut = None
        self.command = None
        self.stop_requested = False

    def build_phase(self):
        self.ap = uvm_analysis_port("ap", self)

    async def run_phase(self):
        self.raise_objection()
        while not self.stop_requested:
            cmd = await self.seq_item_port.get_next_item()

            if cmd.cmd_tr == "Initialize":
                await self.initialize_sequence()
            elif cmd.cmd_tr == "RandomFC":
                for _ in range(15):
                    transaction = await self.randomfc_sequence()
                    self.ap.write(transaction)


            self.seq_item_port.item_done()

        self.drop_objection()

    async def initialize_sequence(self):
        await ClockGen(self.dut)
        await ResetDut(self.dut)
        await SigInitialize(self.dut)
        await SendFC_Idle(self.dut)

    async def randomfc_sequence(self):
        transaction = await SendRandomFC(self.dut)
        return transaction


    def stop(self):
        self.stop_requested = True
```

# PYUVM for FC Controller

## FCMonitor (PYUVM Monitor)

- Purpose
  - Monitors signals from the DUT to capture behavior and ensure correct operation.
- Initialization
  - Configures to monitor specific cycles of interest to capture signal changes.
- Run Phase
  - Monitors changes in signals, especially clk40_out_p_tb, to track DUT behavior.
  - Captures data from the DUT and sends analyzed transactions to the analysis port (ap).
  - Provides detailed insights into signal integrity and performance metrics.

```
class FCMonitor(uvm_component):
    def __init__(self, name, parent, dut, cycles_to_monitor=15*6):
        super().__init__(name, parent)
        self.dut = dut
        self.cycles_to_monitor = cycles_to_monitor
        self.ap = uvm_analysis_port("ap", self)

    def build_phase(self):
        pass

    async def run_phase(self):
        self.raise_objection()
        await RisingEdge(self.dut.clk40_out_p_tb)

        for i in range(self.cycles_to_monitor):
            await RisingEdge(self.dut.clk40_out_p_tb)
```

```
    for signal_name in transaction.keys():
        signal_value = getattr(self.dut, signal_name).value
        if signal_value is not None and signal_value.is_resolvable:
            transaction[signal_name] = signal_value.integer
    self.ap.write(transaction)
    logger.info(f"Transaction: {transaction}")
self.drop_objection()
```

# PYUVM for FC Controller

## FC_Scoreboard(PYUVM Scoreboard)

- Purpose
  - Verifies DUT behavior by comparing expected results with actual outputs.
  - Captures command and result transactions for analysis.
- Connect Phase
  - Connects cmd_get_port and result_get_port to their respective FIFO get exports.
- Check Phase
  - Iterates through transactions to validate correctness.
  - Compares received results against expected behavior based on issued commands.
  - Counts successful checks (check_count) and identifies errors if validation fails.

```
class FCScoreboard(uvm_component):
    def __init__(self, name, parent, dut):
        super().__init__(name, parent)
        self.dut = dut
        self.cmd_fifo = uvm_tlm_analysis_fifo("cmd_fifo", self)
        self.result_fifo = uvm_tlm_analysis_fifo("result_fifo", self)
        self.cmd_get_port = uvm_get_port("cmd_get_port", self)
        self.result_get_port = uvm_get_port("result_get_port", self)
        self.cmd_export = self.cmd_fifo.analysis_export
        self.result_export = self.result_fifo.analysis_export

    def connect_phase(self):
        self.cmd_get_port.connect(self.cmd_fifo.get_export)
        self.result_get_port.connect(self.result_fifo.get_export)
```

```
elif command == "CMD_L1A_CALPULSEEXT":
    check = result["L1A_tb"] & result["CalPulseExt_tb"]
elif command == "CMD_L1A_CALPULSEEXT_PREL1A":
    check = result["CalPulseExt_tb"]
elif command == "CMD_L1A_CALPULSEINT_PREL1A":
    check = result["L1A_tb"] & result["CalPulseInt_tb"]
elif command == "CMD_BCR":
    check = result["BCR_tb"]
```

```
    if check:
        check_count+=1
        check =  0
    else:
        errors.append([command,result,test_count])

print("\n\nAll tests completed! Score %d / %d" %(check_count,test_count))
if errors:
    print("\n\n")
    print(errors)
```

## Conclusion

- Achievement
  - Developed and implemented both testbenches for the IP verification within approximately 6-8 weeks.
- Verification
  - Successfully carried out IP verification with added randomization and regression testing.
- Current Work
  - Coding up coverage to enhance verification metrics.
  - Exploring the concept of using config_db for configuration management, akin to uvm_config_db in UVM.