Cocotb-Pynq: Co-simulating Python+RTL applications targeting Pynq platforms with Cocotb

Gavin Lusby
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada
gdplusby@uwaterloo.ca

Nachiket Kapre
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada
nachiket@uwaterloo.ca

***Abstract*—The AMD Pynq ecosystem fails to provide a seamless way to easily validate functional correctness of RTL designs when part of the application logic runs in Python on the ARM (or x86) host CPU. Application developers must wait for the entire FPGA bitstream generation flow and deploy their code to the FPGA before they confirm the correctness of the Python host code working with the RTL design implemented on the FPGA. In contrast, Cocotb offers a Pythonic framework to test and simulate RTL designs in a variety of cycle-accurate simulators, but lacks easy integration with the Pynq ecosystem. In this paper, we propose Cocotb-Pynq, a framework for co-simulating Python ARM (or x86) host code and RTL/Verilog programs in a single environment. This eliminates the need for bitstream generation prior to co-simulation of Python and RTL components and significantly speeds up design iterations. We rewrite key components of the Pynq ecosystem to be `cocotb`-compatible and offer drop-in solutions for Pynq APIs in Cocotb. Specifically, we rewrite the MMIO and AXI DMA blocks using the Python asyncio library to be compatible with Cocotb emulation. We evaluate our framework on a suite of benchmark programs and quantify their performance. In contrast to bitstream generation times of 10 minutes needed for Pynq devices such as Pynq-Z1 for our small benchmarks with modest frequency targets, a Cocotb-Pynq co-simulation takes 1–2 minutes of runtime even for large designs using the entire chip. The framework will be open-sourced and made available for community contributions and evolution.**

***Index Terms*—Digital Simulation, Verification, HDL**

I. Introduction

Since its introduction to the world in 2017, Pynq [1] has revolutionized the deployment and design of FPGA-accelerated embedded applications. Although originally closely tied to the Jupyter environment, the underlying Pythonic framework is valuable on its own merit and has been widely used in research and development. Before Pynq, embedded developers had to write low-level FPGA device drivers for FPGA programming and data movement while also wrangling custom operating system configurations and compilations before they could deploy application split across hardware and software. Pynq abstracted away a lot of this developer tedium in a standardized way across a range of embedded FPGA boards starting with the Pynq-Z1, and growing to include a range of AMD MPSoCs. They expanded range to even include Alveo datacenter cards like the U280 with x86 host CPUs, and have continued to support and evolve the framework.

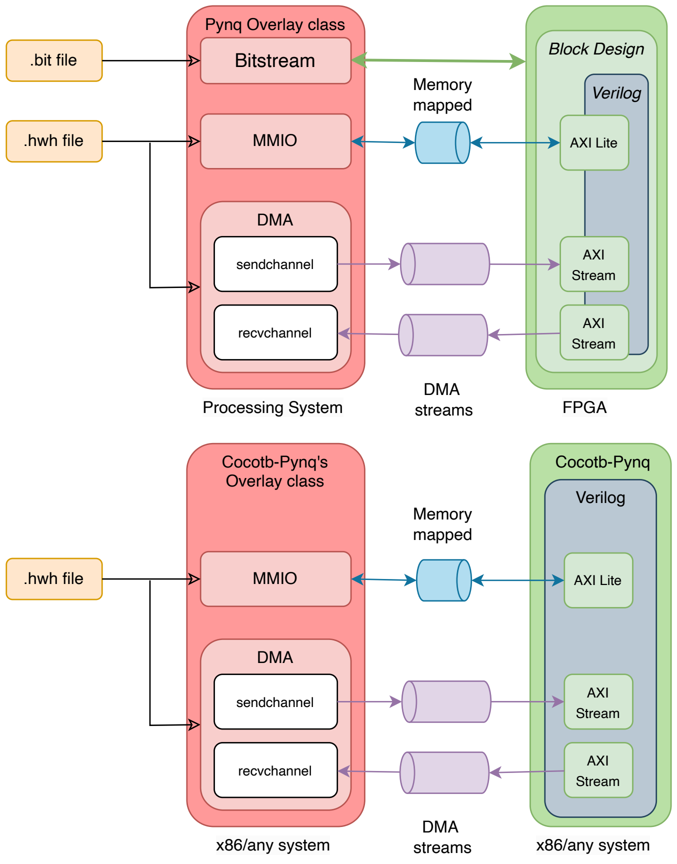

Fig. 1. System-level runtime view of Pynq and Cocotb-Pynq framedworks

What are some specific problems with Pynq? Despite all the obvious benefits of standardized interfaces and abstraction, Pynq requires developers to co-develop applications in RTL (or HLS) and Python. While it is possible to write careful testbenches to rigorously evaluate and analyze your RTL behavior, Pynq wraps your RTL in a Block Design before deploying to the board. A system-level simulation including the processing system (CPU) and the FPGA may be possible, but it would be much too slow to be useful. Furthermore, this will require modeling complex interactions between the ARM CPUs, the DRAM subsystem, the FPGA interfaces for embedded SoCs, and PCIe interfaces for Alveo boards. An alternative simulation approach that models your RTL and the

Python host code together would be desirable. Cocotb [2] may be the answer.

`cocotb` [2] is a coroutine-based Pythonic testing environment for RTL designs. Unlike traditional RTL simulators, `cocotb` uses Python's `asyncio` support to implement testbenches as Python coroutines that interact with your RTL code in complex, realistic ways. This gives programmers the ability to describe their tests using high-level Python language without sacrificing the cycle-by-cycle interactivity necessary for RTL abstraction of hardware designs. The RTL itself can be compiled using existing simulators like `iverilog` [3], `verilator` [4] among others, although notably lacks support for `xsim` due to lack of VPI support in Vivado.

Can we combine the benefits of Pynq for FPGA deployment with those of `cocotb` for fast, expressive simulations? Cocotb-Pynq, shown in Figure 1, answers this question affirmatively. With minimal cosmetic modifications to the Pynq scripts, we can reuse the same Pynq host code as a `cocotb` coroutine and model MMIO and DMA operations in a simple manner. We rewrite the key Pynq libraries necessary for canonical application use-cases to show how to make `asyncio` coroutines interoperate with Pynq Python code.

In the absence of such an environment, Pynq users are forced to wait until bitstream generation and in-system deployment to test whether their Python host code and FPGA RTL logic interoperates correctly as intended. This has often led to wasted developer time due to bugs at the boundary of software and hardware interfaces. Specifically, in the author's experience, bugs related to data packing, data alignment, and incorrect clock configuration in the Pynq board tcl files has required complex debugging methods. Often, developers are forced to reason about correctness through expensive, and time-consuming ILA (integrated logic analyzer) approach only to realize the bugs were merely in the Python code. Cocotb-Pynq hopes to avoid these wasted development times.

In this paper, we make the following contributions:

- Design and engineering of Cocotb-Pynq framework to allow Pynq Python host code to be co-simulated with RTL designs.
- Demonstration of correct, and fast operation of representative examples such as simple polynomial evaluation, and matrix multiplication involving the use of both MMIO and AXI DMA operations.
- Performance evaluation and characterization of the framework to quantify the benefits of the co-simulation environment over existing approaches on Pynq-Z1 board.

II. Cocotb-Pynq Framework

The Cocotb-Pynq framework consists of a few components, best understood by looking at Figure 1. In the conventional Pynq-based flow (shown in the upper-half of the figure), the Pynq runtime executes on the embedded ARM CPU (or x86 for Alveo), and is responsible for (1) configuring the FPGA bitstream, (2) managing memory-mapped IO (MMIO) and DMA operations, and (3) providing a way for users to express application software logic in Python. The Cocotb-Pynq framework (shown in the lower-half of the figure) looks remarkably similar, yet substantially different.

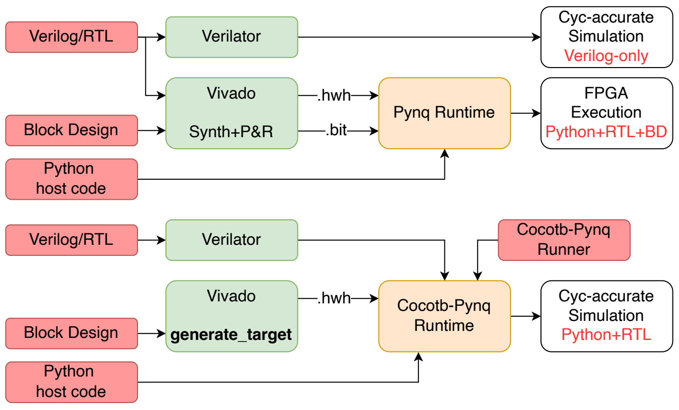

Fig. 2. Compile/Runtime flow of Pynq and Cocotb-Pynq frameworks

- Cocotb-Pynq does not require you to generate a bitstream. If you had to generate an entire FPGA bitstream, there would no longer be any speedup in the simulation flow.
- In fact, Cocotb-Pynq does not even require you to have silicon to do any co-simulation! Cocotb-Pynq can run on any system (x86 or ARM) that can install `cocotb`, and does not require access to any FPGA device.
- Cocotb-Pynq provides Cocotb-compatible implementation of the MMIO and DMA APIs. These are identical to Pynq APIs so Pynq programmers will note no difference in expression and expected execution.

The compilation and runtime flow is also significantly different as shown in Figure 2. As you can see, the traditional Pynq flow (shown in the upper-half of the figure) has a simulation path that takes the Verilog/RTL code and executes it through a simulator such as Verilator to confirm cycle-accurate correctness of the digital design. To deploy the code to an FPGA, you have to wrap it in a block design so the processing system can interface with the RTL as part of an embedded system-on-chip (for ARM CPUs) or PCIe-connected accelerator interface (for x86 CPUs). Regardless of the target, we must use Vivado to generate a bitstream `.bit` and hardware description file `.hwh`. The `.bit` bitstream is downloaded to the FPGA using appropriate configuration drivers for the target. The `.hwh` file is read by the Pynq runtime to determine AXI DMA and MMIO mappings for runtime interaction with the configured FPGA. Thus, the only way to verify the host Python code with the RTL is directly on the FPGA, unless you want to do a much slower cycle-accurate Processing System + Programmable Logic simulation using `qemu` or some other exotic technique.

In contrast, the Cocotb-Pynq flow does not run Vivado compilation. We only run the `generate_target` command that takes the Block design file that specifies connectivity between the CPU and FPGA components. This is enough to generate the `.hwh` file which is essential for mapping the AXI DMA and MMIO operations in the Pynq host code

Cocotb Code

```
import cocotb
from cocotb.triggers import RisingEdge
import random

@cocotb.test
async def main(dut):

    # Reset the DUT
    dut.rst.value = 1
    await RisingEdge(dut.clk)
    dut.rst.value = 0

    await axi_lite_write(0x10, 1) # write a
    await axi_lite_write(0x18, 2) # write b
    await axi_lite_write(0x20, 3) # write c

    in_buff = [random.randint(0, 100)
               for _ in range(5)]
    out_buff = []

    await cocotb.start(
        axi_stream_write(dut, in_buff))
    await axi_stream_read(dut, out_buff)

    print("Answer:")
    print(out_buff)

async def axi_lite_write(addr, data):
    ... #manual implementation by user

async def axi_stream_write(addr, in_buff):
    ... #manual implementation by user

async def axi_stream_read(addr, out_buff):
    ... #manual implementation by user
```

Pynq Code

```
from pynq import Overlay
from pynq import MMIO
from pynq import allocate
from pynq import PL


def main():
    # program FPGA
    overlay = Overlay('./fpga.bit')

    # write a, b, c
    mmio = MMIO(0x43C10000, 0x1000)
    mmio.write(0x10, 1) # write a
    mmio.write(0x18, 2) # write b
    mmio.write(0x20, 3) # write c

    # stream inputs/outputs over DMA
    in_buff = allocate(shape=(5,),
                dtype=np.uint32)
    out_buff = allocate(shape=(5,),
                dtype=np.uint32)

    in_buff[:] = np.random.randint(0, 100,
        size=in_buff.shape, dtype=np.uint32)

    recv=overlay.poly.axi_dma.recvchannel;
    send=overlay.poly.axi_dma.sendchannel;

    recv.transfer(out_buff)
    send.transfer(in_buff)
    send.wait()
    recv.wait()

    print("Answer:")
    print(out_buff)

if __name__ == "__main__":
    main()
```

Cocotb-Pynq Code

```
import cocotbpynq
from cocotbpynq import Overlay
from cocotbpynq import MMIO
from cocotbpynq import allocate
from cocotbpynq import PL

@cocotbpynq.synctest
def main(dut):
    # program FPGA
    overlay = Overlay('./fpga.bit')

    # write a, b, c
    mmio = MMIO(0x43C10000, 0x1000)
    mmio.write(0x10, 1) # write a
    mmio.write(0x18, 2) # write b
    mmio.write(0x20, 3) # write c

    # stream inputs/outputs over DMA
    in_buff = allocate(shape=(5,),
                dtype=np.uint32)
    out_buff = allocate(shape=(5,),
                dtype=np.uint32)

    in_buff[:] = np.random.randint(0, 100,
       size=in_buff.shape, dtype=np.uint32)

    recv=overlay.poly.axi_dma.recvchannel;
    send=overlay.poly.axi_dma.sendchannel;

    recv.transfer(out_buff)
    send.transfer(in_buff)
    send.wait()
    recv.wait()

    print("Answer:")
    print(out_buff)
```

TABLE I
COMPARISON OF COCOTB, PYNQ, AND COCOTB-PYNQ.

with the FPGA AXI ports. We still need to supply a Cocotb runner like any Cocotb simulation but that is a generic wrapper that can be reused across other projects with minimal if any modifications. Thus, the Cocotb-Pynq runtime will combine the AXI DMA and MMIO information from `.hwh` with the Verilator-compiled cycle-accurate simulation executable for the RTL and the Pynq Python host code `.py` running as a testbench coroutine. This strategic assembly of components lies at the heart of Cocotb-Pynq and is what makes it fast as well as accurate.

Dealing with asynchronous waits: The Cocotb-Pynq MMIO and DMA APIs were reasonably easy to re-implement for functional correctness as they naturally match the asynchronous computing model of `cocotb`. However, our initial implementations required the Pynq host code to have **await** statements (asynchronous waits) to ensure compatibility. This forced the host code to deviate too different from its natural expression. Luckily, this was resolved when we discovered how to wrap synchronous functions as `@cocotb.function` to remove the asynchronous waits.

MMIO and AXI DMA Timing Model: When implementing AXI handshakes, we had to carefully order the AXI transactions to ensure full throughput operation. Any bubbles or stalls should be purely from the DUT and not the software wrapper. The adjacent recurring code fragment was essential to ensuring the timing behavior models AXI operation ordering. The key here is the insertion of `ReadOnly()` guard to force values to stabilize before further operations are performed. This results in MMIO writes taking 3 cycles, reads taking 2 cycles, and DMA operations running back-to-back at full throughput. A custom implementation for AXI handshakes was chosen over `cocotbext-axi` [5] to keep the package lightweight.

```
await ReadOnly()
rdy = self.cpbus.AWREADY.value
while(periph_addr_ready == 0b0):
    await RisingEdge(self.cpdut.clk)
    await ReadOnly()
    rdy = self.cpbus.AWREADY.value
await RisingEdge(self.cpdut.clk)
```

A. Code Snippets

To better understand how we unify `cocotb` and Pynq, we show how to create test harnesses for a simple Verilog 1-cycle implementation of a polynomial $a \cdot x^2 + b \cdot x + c$ in Table I. The assumption is that a, b, and c are scalar constants that are loaded at start of execution. The input x is then streamed continuously while output y is available a cycle later. The interactions are mapped to MMIO (for scalars) and AXI DMA for the streams. As you can see,

Design	Verilator (Ryzen 9)		Pynq (Pynq-Z1)			Cocotb-Pynq (Ryzen 9)			
	Verilator Compile $t_{compile}$	Testcase Runtime t_{swexec}	Bitstream Gen. Time t_{vivado}	Pynq Overhead t_{pynqoh}	Testcase Runtime $t_{fpgaexec}$	`.hwh` generation t_{hwh}	Verilator compile $t_{compile}$	Cocotb Overhead t_{runner}	Testcase Runtime t_{swexec}
polynomial	6.1 s	0.006 s	435 s	9.1 s	3.01 s	31.8 s	8 s	1.4 s	0.02 s
mat_mul 4x4	8.4 s	0.23 s	356 s	8.8 s	3.07 s	31.8 s	11.7 s	1.4 s	0.08 s
mat_mul 12x12	34.7 s	0.24 s	570 s	9.6 s	3.97 s	31.8 s	30.2 s	1.4 s	0.56 s
mat_mul 16x16	37.4 s	0.33 s	698 s	9.0 s	4.75 s	31.8 s	52.5 s	1.6 s	1.19 s

Fig. 3. Comparing execution times of the various simulations and FPGA executions.

the three code blocks looks very similar. `cocotb` tests interact with the design on a cycle-by-cycle basis dealing with `reset` and then handling the AXI protocol interactions for `x` and `y`. Both Pynq and Cocotb-Pynq explicitly support MMIO transactions and DMA transfers. They do not make the programmer reason about cycle-by-cycle operation of the MMIO and AXI DMA interfaces. Specifically, the Pynq FPGA execution behaves exactly the same way by abstracting the AXI protocol handling, and DMA IP management logic. The key takeaway is that the Cocotb-Pynq code looks pretty much identical to the Pynq code, barring the imports and the function decoration `@cocotbpynq.synctest`. This decorator, which is supplied by the Cocotb-Pynq framework, wraps a synchronous function as a cocotb test. This allows the code to call asynchronous `cocotb` functions that are wrapped with decorates `@cocotb.function` such as DMA wait and MMIO read/write.

III. Experimental Evaluation

We now evaluate the performance (runtime) of the Cocotb-Pynq framework compared to the traditional Pynq development flow. Our tcl currently targets the Pynq-Z1 FPGA board but can support other Pynq-compatible boards as long as it uses compatible AXI-DMA IP and MMIO logic. We consider the simple polynomial example as a small microbenchmark and then consider an `int8` systolic array with varying sizes as a scaling study to evaluate time trends as a function of RTL complexity. All runtime measurements of tests are done across 100 experimental runs and the average runtime is reported. We use a Ryzen9-5900X 12-core Ubuntu server with 128GB RAM for our mapping and testing experiments with Verilator and Cocotb-Pynq. We use the Pynq-Z1 board for our Pynq experiments.

We consider three experiments:

- **Verilator**: We will evaluate the runtime of software simulation of RTL code using `verilator` simulator. We expect runtime here to be $t_{verilator} = t_{compile} + t_{swexec}$ where the compile time ($t_{compile}$) is a one-time cost while software execution (t_{swexec}) scales with test complexity.
- **Pynq**: For the traditional Pynq flow, we will measure the runtime of FPGA bitstream generation (t_{vivado}) and FPGA execution. The runtime expression is $t_{pynq} = t_{vivado} + t_{pynqoh} + t_{fpgaexec}$. FPGA execution involves some constant overhead (t_{pynqoh}) from setting up the script (*e.g.* `import` statements) as well as time from actual execution in silicon ($t_{fpgaexec}$).
- **Cocotb-Pynq**: Finally, we will measure the runtime of the Cocotb-Pynq framework, using Vivado to generate `.hwh` file (t_{hwh}) and `verilator` to compile ($t_{compile}$) a software model of the RTL for evaluation. Here, we expect the runtime to be more complicated $t_{cocotbpynq} = t_{hwh} + t_{compile} + t_{runner} + t_{swexec}$. Cocotb runner's overhead, t_{runner}, is similar to Pynq's overheads.

In Table 3, we compare the execution runtime profiles of different benchmarks and different conditions. For the simple polynomial example, we note that the Verilator simulation takes ≈6 s while traditional Pynq FPGA execution took ≈446 s. The Cocotb-Pynq flow took ≈40 s with the bulk of the overhead due to `.hwh` file generation needs. Note that this only needs to be done once for a given design, and rerun when the CPU-FPGA interface changes (*e.g.* a new DMA channel is added) which is very infrequent if at all. If the system-level interface is stable, the execution runtime is then similar to the Verilator runtime barring a 1.4 s Cocotb runner overhead. When we scale the design to larger systolic array design, we mainly see the `verilator` compile time increase significantly, while the testcase runtime is negligible in comparison. However, the FPGA bitstream generation time increases dramatically to ≈700 s with design complexity and will be the key stumbling block for testing. Unsurprisingly `.hwh` file generation time stays stable ≈30 s as the block design complexity is stable.

IV. Conclusions

We present Cocotb-Pynq, a simulation framework that allows embedded FPGA developers to co-simulate their RTL digital designs with the Python host code in a single, fast environment. This can replace the traditional Pynq design flow that requires deployment to the FPGA to test the digital circuit with the Python host code to verify their overall application correctness. This leads to wasteful design iterations as the lack of software simulation visibility hampers the bug discovery flow. We show how to make Pynq code compatible with `cocotb`'s asynchronous coroutines framework by avoiding bitstream generation and speeding up the simulation flow. In our runtime characterization experiments, we note that barring one-time 30 s `.hwh` file generation runtime and modest 1 s `cocotb` runner overheads, Cocotb-Pynq performance is similar to Verilator software execution.

References

[1] Benjamin John Rosser. 2018. Cocotb: a Python-based digital logic verification framework. In Micro-electronics Section seminar. CERN, Geneva, Switzerland.

[2] Xilinx. 2017. PYNQ: Python productivity for Zynq. In International conference on Field Programmable Logic.

[3] Stephen Williams and Michael Baxter. 2002. Icarus verilog: open-source verilog more than a year later. Linux Journal 2002, 99 (2002), 3.

[4] Wilson Snyder. 2004. Verilator and systemperl. In North American SystemC Users' Group, Design Automation Conference, 122–148.'

[5] Alex Forencich. 2020. cocotbext-axi. Github, Available: https://github.com/alexforencich/cocotbext-axi/