Cocotb-Pynq

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

Gavin Lusby, Dr. Nachiket Kapre

Introduction to Cocotb-Pynq

Introduction to Cocotb-Pynq

- Co-verify Python host code and RTL in the same environment!

Introduction to Cocotb-Pynq

- Co-verify Python host code and RTL in the same environment!
- Can be on the order of 10-50x faster*

PYNQ - Productivity for ZYNQ

PYNQ - Productivity for ZYNQ

+ Ease of prototyping hardware accelerators needing to interact with CPU

PYNQ - Productivity for ZYNQ

+ Ease of prototyping hardware accelerators needing to interact with CPU
+ Access to breadth of python software libraries

PYNQ - Productivity for ZYNQ

+ Ease of prototyping hardware accelerators needing to interact with CPU
+ Access to breadth of python software libraries
+ Abstracts logic into software-like libraries using *overlays*

PYNQ - Productivity for ZYNQ

+ Ease of prototyping hardware accelerators needing to interact with CPU
+ Access to breadth of python software libraries
+ Abstracts logic into software-like libraries using *overlays*

− Very difficult to debug faulty interactions between CPU and logic

PYNQ - Productivity for ZYNQ

- + Ease of prototyping hardware accelerators needing to interact with CPU
- + Access to breadth of python software libraries
- + Abstracts logic into software-like libraries using *overlays*

- − Very difficult to debug faulty interactions between CPU and logic
- − Full bitstream re-generation can take hours when RTL is modified

PYNQ - Productivity for ZYNQ

+ Ease of prototyping hardware accelerators needing to interact with CPU
+ Access to breadth of python software libraries
+ Abstracts logic into software-like libraries using *overlays*

− Very difficult to debug faulty interactions between CPU and logic
− Full bitstream re-generation can take hours when RTL is modified
− Requires physical access to ZYNQ board to do any end-to-end testing

cocotb - Python Verification Framework

cocotb - Python Verification Framework

+ Simulator agnostic

cocotb - Python Verification Framework

+ Simulator agnostic
+ Write your testbench in python(Like PYNQ, hint hint)

cocotb - Python Verification Framework

+ Simulator agnostic
+ Write your testbench in python(Like PYNQ, hint hint)

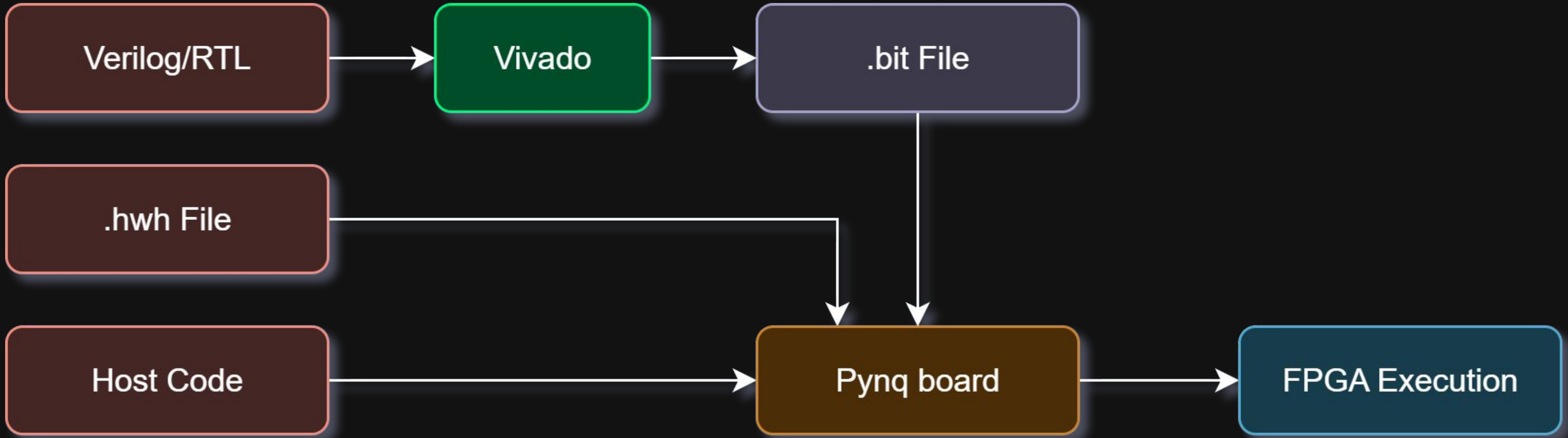
− Can only verify PL/RTL logic (implicitly)

cocotb - Python Verification Framework

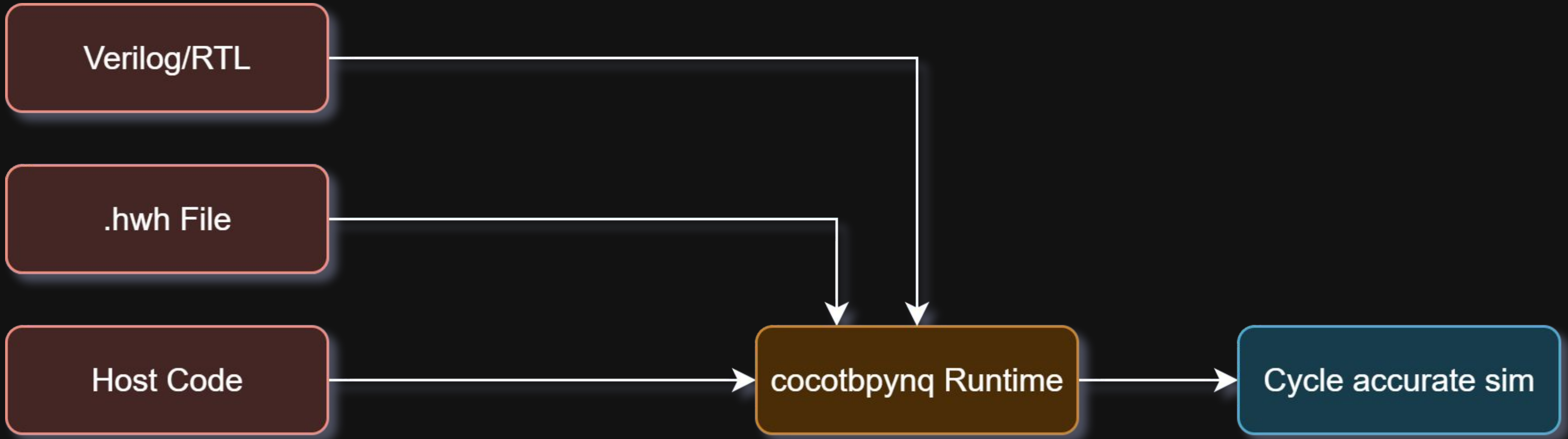
- + Simulator agnostic
- + Write your testbench in python(Like PYNQ, hint hint)

- − Can only verify PL/RTL logic (implicitly)
- − Learning curve

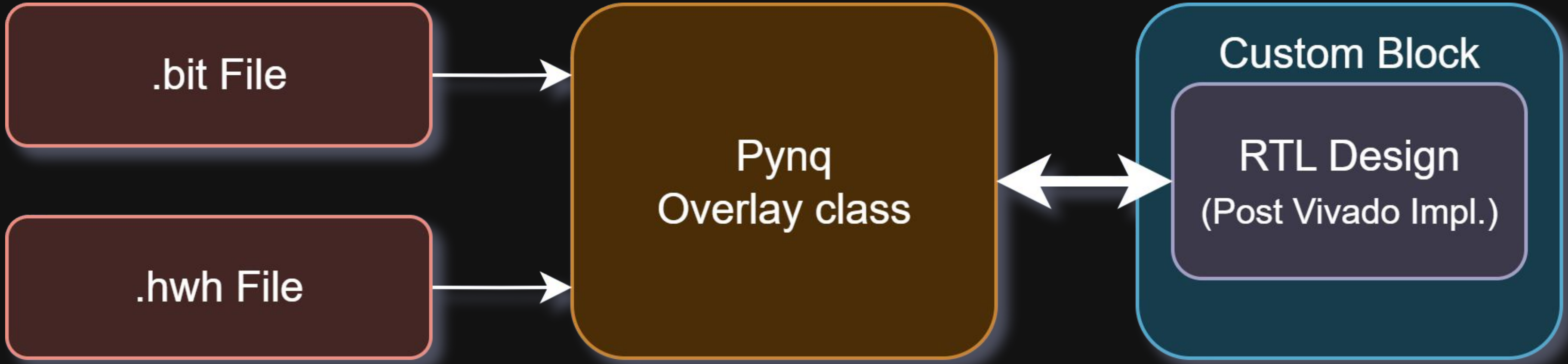
Pynq Compilation Flow



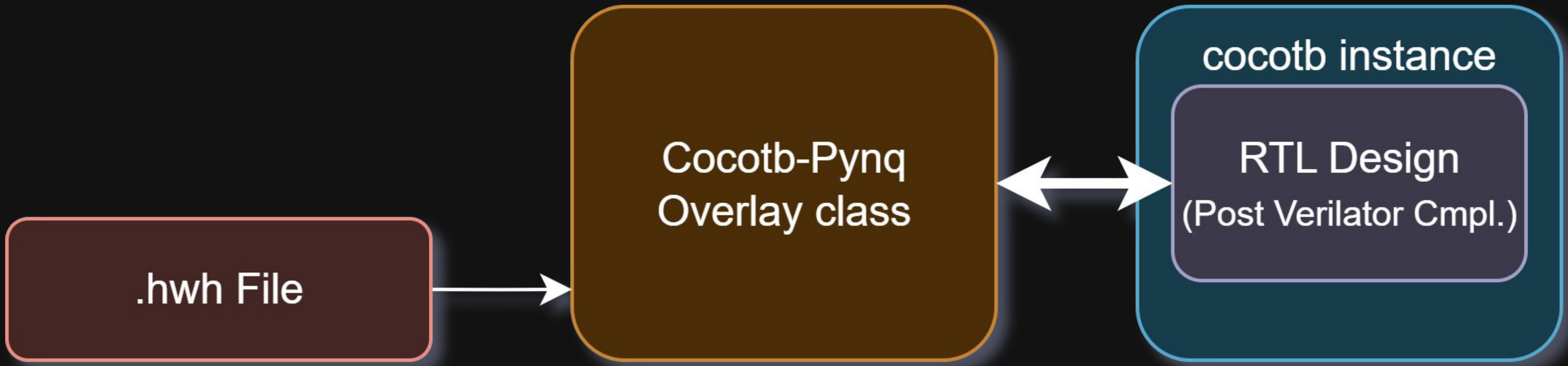
Cocotb-Pynq Compilation Flow



Pynq Runtime

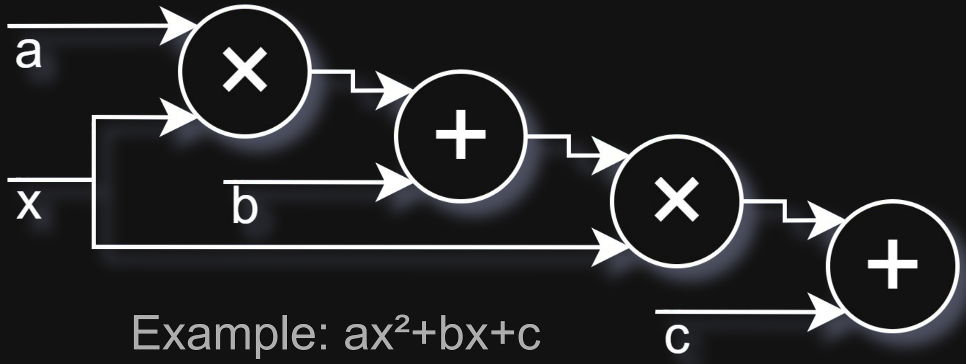


Cocotb-Pynq Runtime

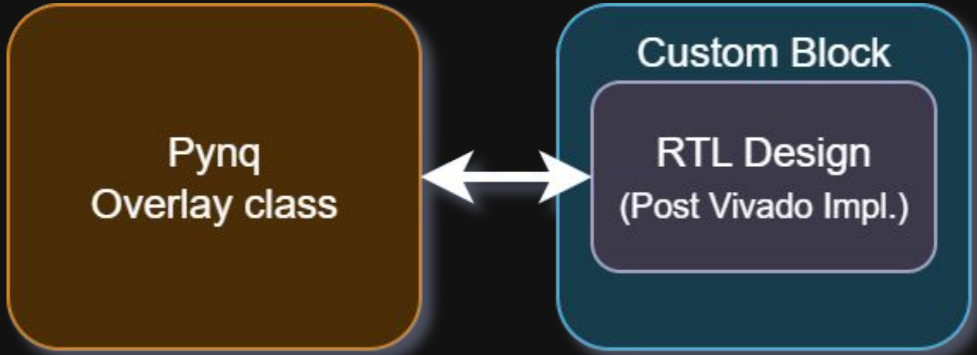


```python
from pynq import Overlay
from pynq import MMIO
from pynq import allocate
from pynq import PL



def main():
    # program FPGA with overlay
    overlay = Overlay('./fpga.bit')
    # write a, b, c
    mmio = MMIO()
    mmio.write(ADDR_A, 1)
    mmio.write(ADDR_B, 2)
    mmio.write(ADDR_C, 3)
    # declare DMA channels/buffers
    ...
    # carry out transaction
    recv.transfer(out_buff)
    send.transfer(in_buff)
    send.wait()
    recv.wait()
    print("Answer:", out_buff)
```

Original Host Code

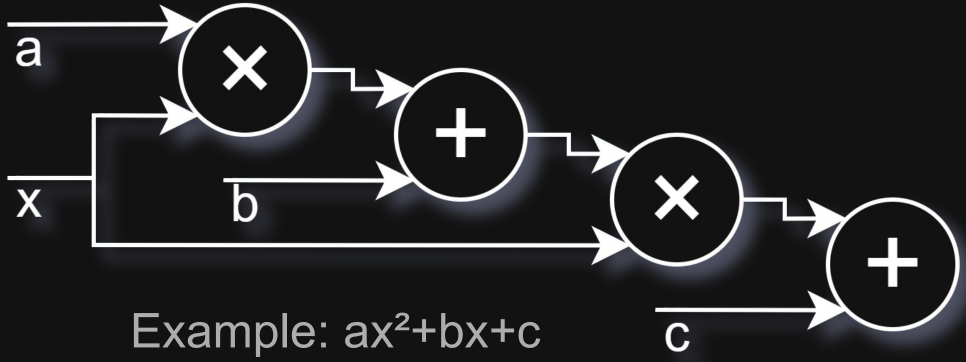


Example: ax^2+bx+c

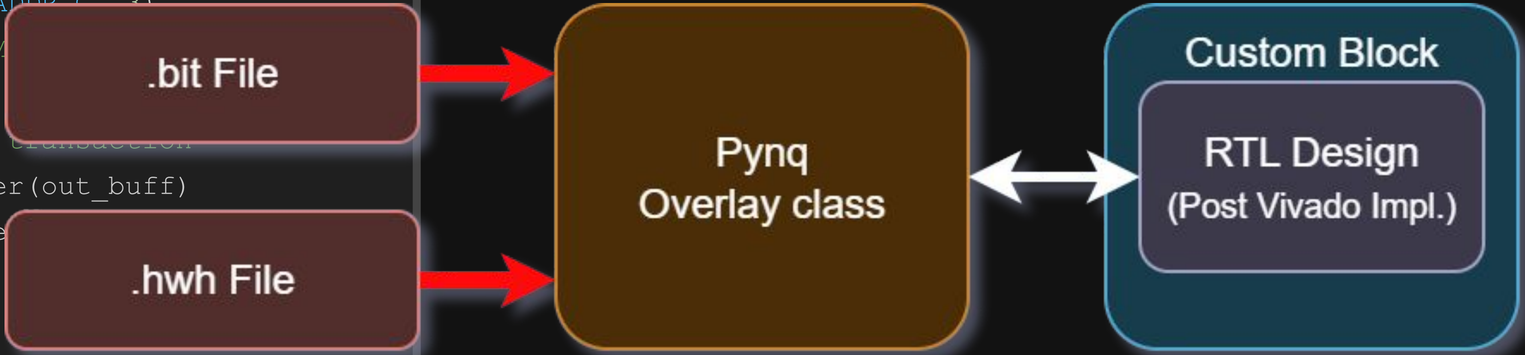


Original Host Code

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


def main():
    # program FPGA with overlay
    overlay = Overlay('./fpga.bit')
    # write a, b, c
    mmio = MMIO()
    mmio.write(ADDR_A, 1)
    mmio.write(ADDR_B, 2)
    mmio.write(ADDR_C, 3)
    # declare DM
    ...
    # carry out transaction
    recv.transfer(out_buff)
    send.transfe
    send.wait()
    recv.wait()
    print("Answer:", out_buff)
```

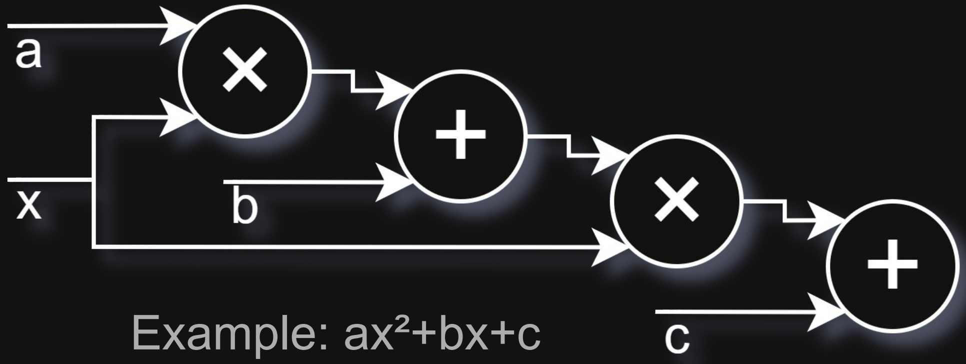


Example: ax^2+bx+c

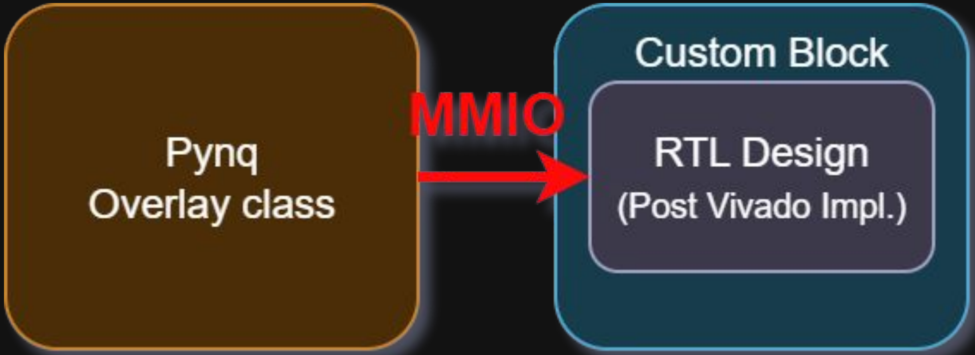


```
from pynq import Overlay
from pynq import MMIO
from pynq import allocate
from pynq import PL


def main():
    # program FPGA with overlay
    overlay = Overlay('./fpga.bit')
    # write a, b, c
    mmio = MMIO()
    mmio.write(ADDR_A, 1)
    mmio.write(ADDR_B, 2)
    mmio.write(ADDR_C, 3)
    # declare DMA channels/buffers
    ...
    # carry out transaction
    recv.transfer(out_buff)
    send.transfer(in_buff)
    send.wait()
    recv.wait()
    print("Answer:", out_buff)
```

Original Host Code

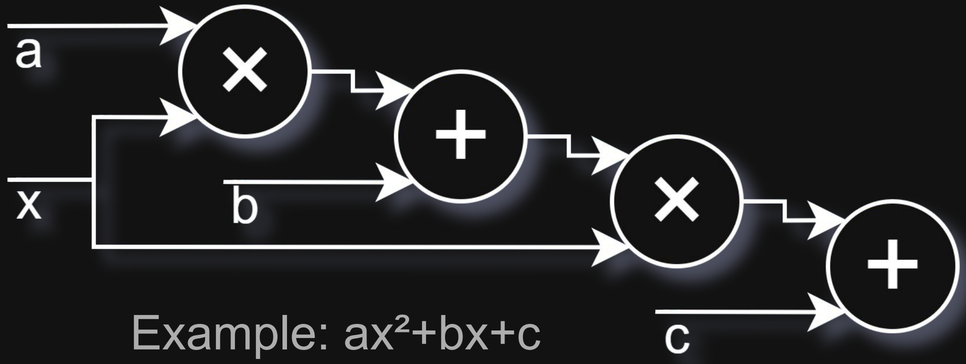


Example: ax^2+bx+c

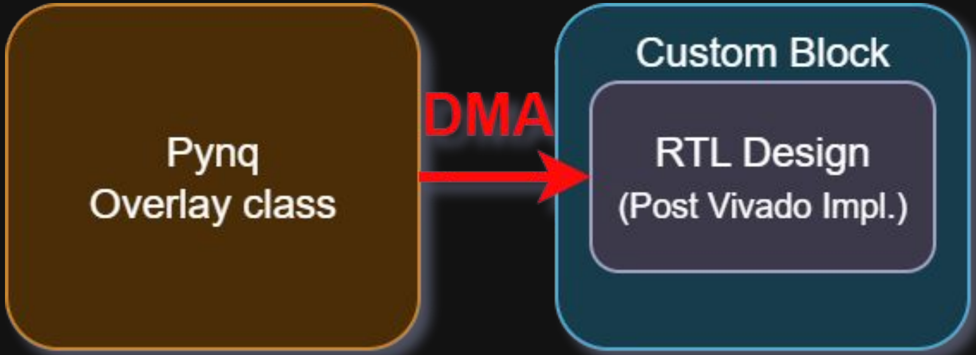


```
from pynq import Overlay
from pynq import MMIO
from pynq import allocate
from pynq import PL


def main():
    # program FPGA with overlay
    overlay = Overlay('./fpga.bit')
    # write a, b, c
    mmio = MMIO()
    mmio.write(ADDR_A, 1)
    mmio.write(ADDR_B, 2)
    mmio.write(ADDR_C, 3)
    # declare DMA channels/buffers
    ...
    # carry out transaction
    recv.transfer(out_buff)
    send.transfer(in_buff)
    send.wait()
    recv.wait()
    print("Answer:", out_buff)
```

Original Host Code

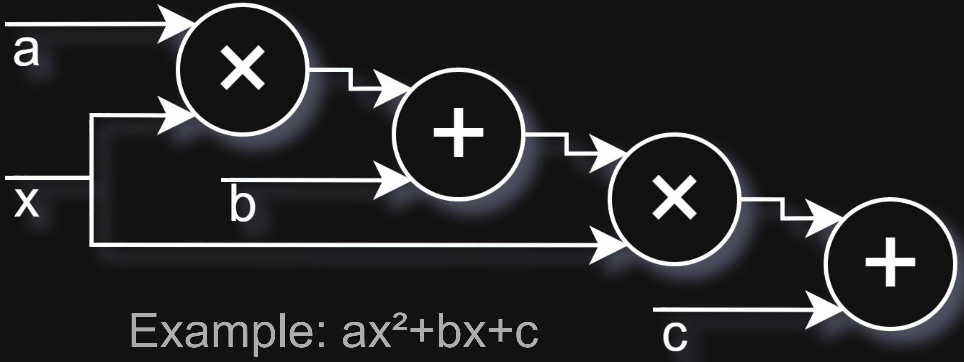


Example: ax^2+bx+c

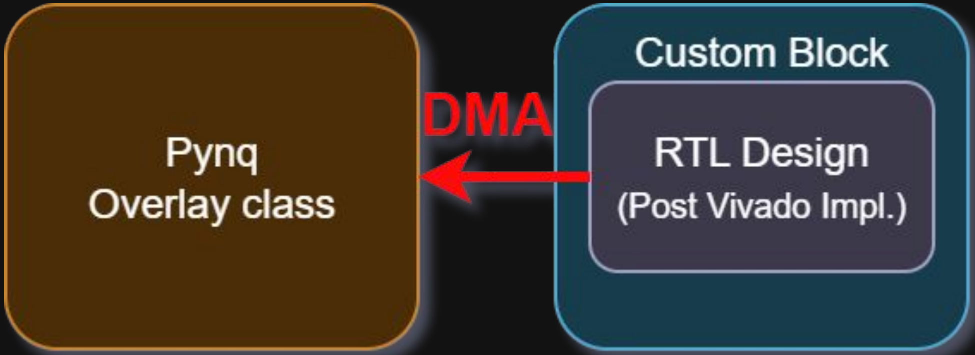


Original Host Code

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



def main():
    # program FPGA with overlay
    overlay = Overlay('./fpga.bit')
    # write a, b, c
    mmio = MMIO()
    mmio.write(ADDR_A, 1)
    mmio.write(ADDR_B, 2)
    mmio.write(ADDR_C, 3)
    # declare DMA channels/buffers
    ...
    # carry out transaction
    recv.transfer(out_buff)
    send.transfer(in_buff)
    send.wait()
    recv.wait()
    print("Answer:", out_buff)
```

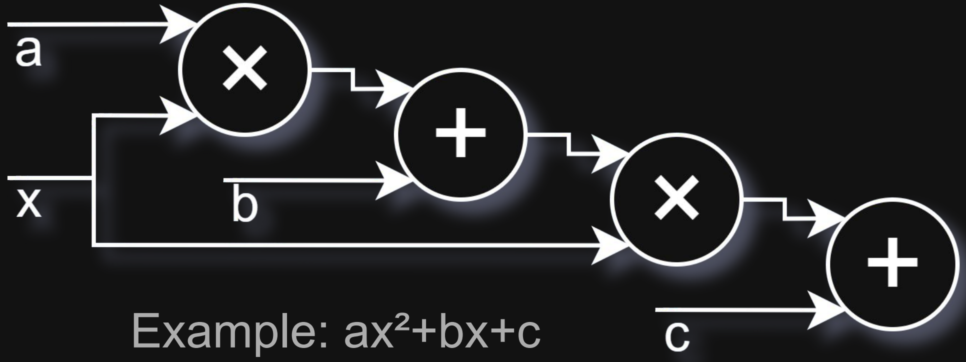


Example: ax^2+bx+c

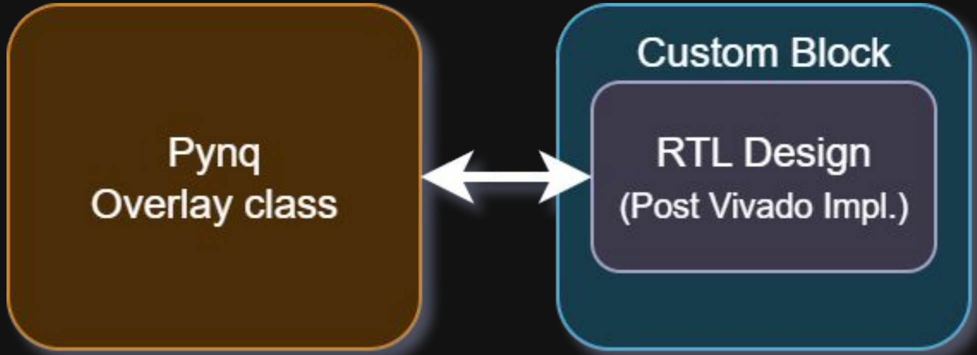


```
from pynq import Overlay
from pynq import MMIO
from pynq import allocate
from pynq import PL



def main():
    # program FPGA with overlay
    overlay = Overlay('./fpga.bit')
    # write a, b, c
    mmio = MMIO()
    mmio.write(ADDR_A, 1)
    mmio.write(ADDR_B, 2)
    mmio.write(ADDR_C, 3)
    # declare DMA channels/buffers
    ...
    # carry out transaction
    recv.transfer(out_buff)
    send.transfer(in_buff)
    send.wait()
    recv.wait()
    print("Answer:", out_buff)
```

Original Host Code

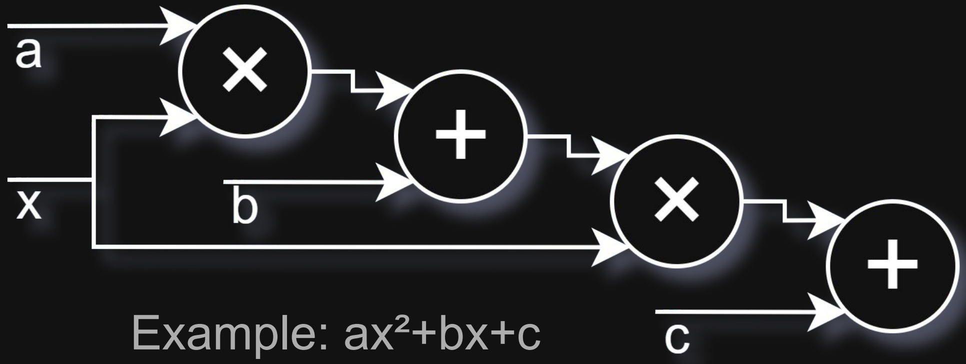


Example: ax^2+bx+c

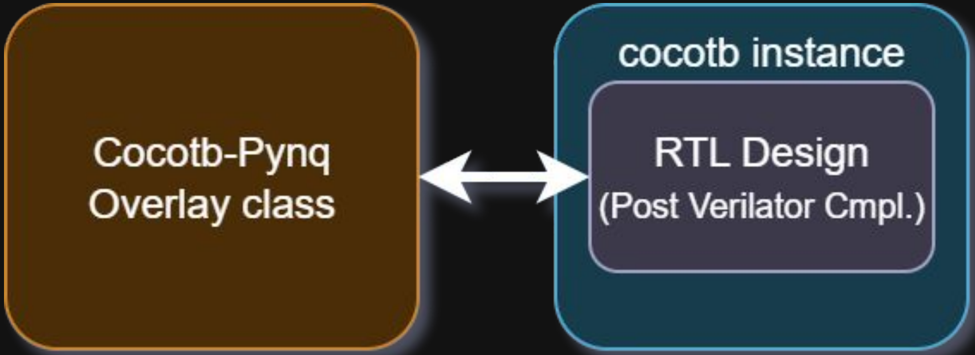


Modified Host Code

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

@cocotbpynq.synctest
def main():
    # program FPGA with overlay
    overlay = Overlay('./fpga.bit')
    # write a, b, c
    mmio = MMIO()
    mmio.write(ADDR_A, 1)
    mmio.write(ADDR_B, 2)
    mmio.write(ADDR_C, 3)
    # declare DMA channels/buffers
    ...
    # carry out transaction
    recv.transfer(out_buff)
    send.transfer(in_buff)
    send.wait()
    recv.wait()
    print("Answer:", out_buff)
```

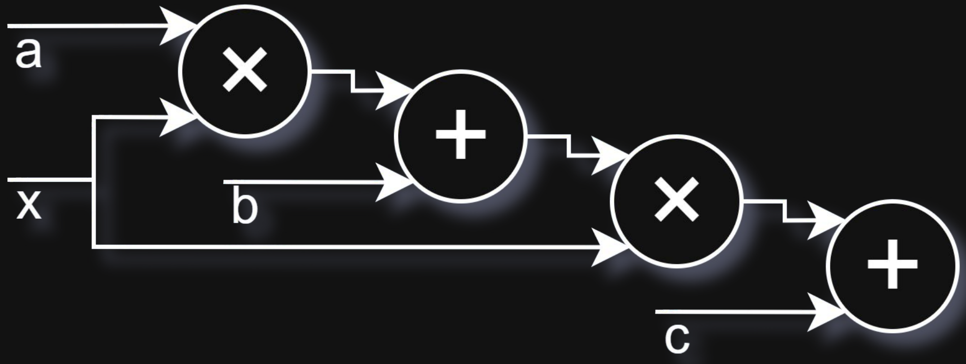


Example: ax^2+bx+c



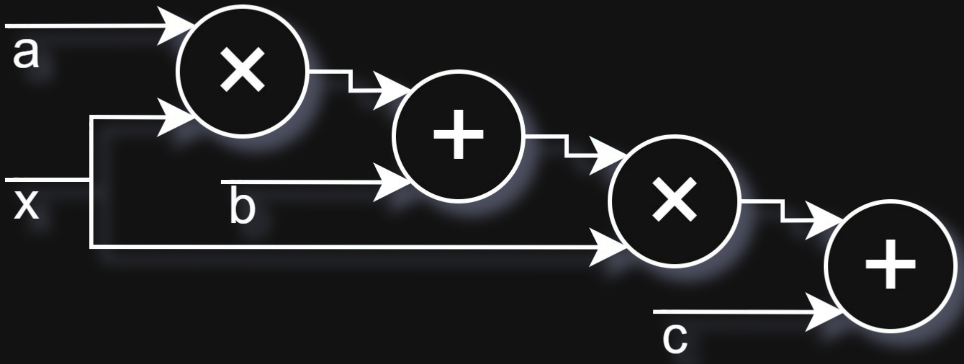
Breakdown (polynomial)

Pynq-Z1		Cocotb-Pynq	
Step	Execution Time	Step	Execution Time



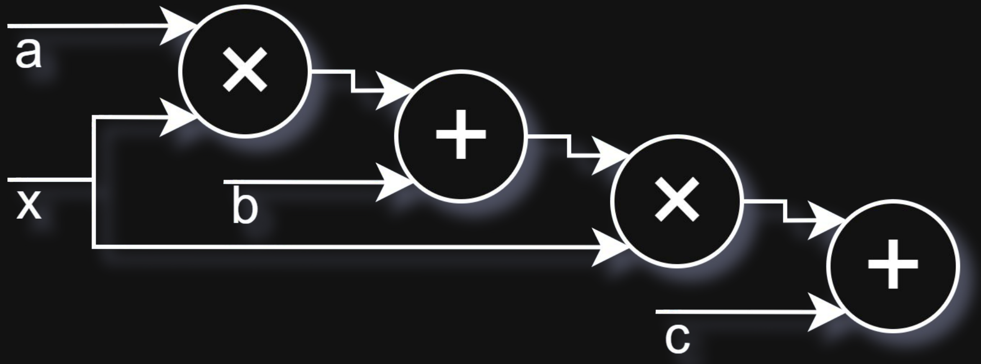
Breakdown (polynomial)

Pynq-Z1		Cocotb-Pynq	
Step	Execution Time	Step	Execution Time
Bitstream Gen. Time	435s	Verilator Compile	8.0s



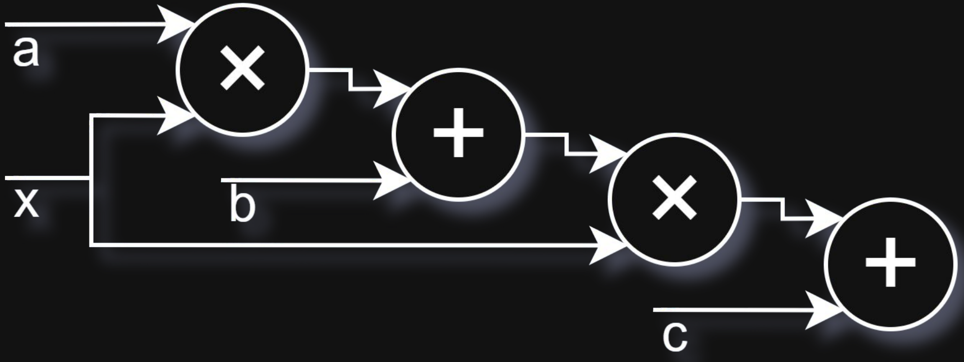
Breakdown (polynomial)

Pynq-Z1		Cocotb-Pynq	
Step	Execution Time	Step	Execution Time
Bitstream Gen. Time	435s	Verilator Compile	8.0s
Pynq Overhead	9.1s	Cocotb Overhead	1.5s



Breakdown (polynomial)

Pynq-Z1		Cocotb-Pynq	
Step	Execution Time	Step	Execution Time
Bitstream Gen. Time	435s	Verilator Compile	8.0s
Pynq Overhead	9.1s	Cocotb Overhead	1.5s
Test	3.0s	Test	0.02s



Timing Results

Design	Pynq-Z1			Cocotb-Pynq (Ryzen 9)			
	Bitstream Gen. Time	Pynq Overhead	Test Runtime	Verilator Compile	Cocotb Overhead	Test Runtime	Speedup
polynomial	435s	~9.1s	3.0s	8.0s	~1.5s	0.02s	47x
mat_mul_4x4	356s		3.1s	11.7s		0.08s	28x
mat_mul_12x12	570s		4.0s	30.2s		0.56s	18x
mat_mul_16x16	698s		4.8s	52.5s		1.19s	13x

*mat_mul_16x16 is on the larger end of what can fit on Pynq-Z1

Conclusion

Speeding up co-verification of Python host code and RTL in same environment

Check out our repository:

https://github.com/watcag/cocotb-pynq

Or download & test polynomial with

```
pip install cocotbpynq
python -m cocotbpynq.sample
```

(Needs Verilator Installed)