基于Python的“RISC-V+通用AI”SoC框架PyYard

赖晓铮 陈若晖 莫国艺

目录 | CONTENT

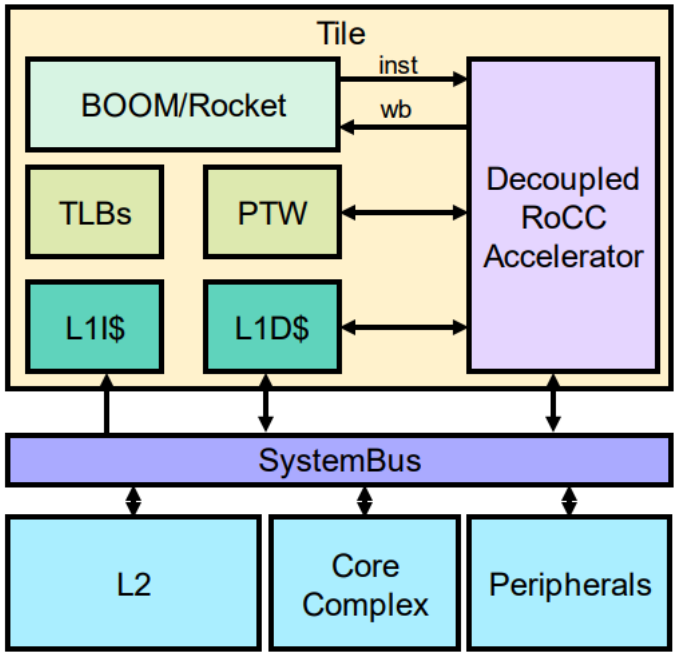
1 PyYard概述
2 PyChip框架
3 VTA体系结构
4 PyHCL与算法设计

PyYard vs ChipYard

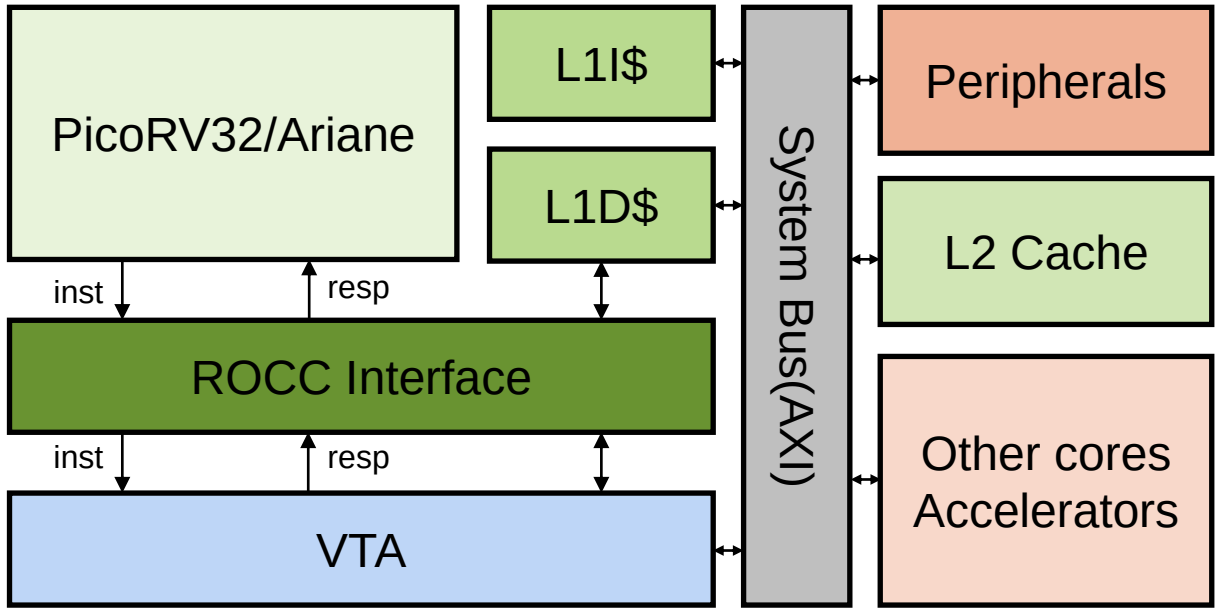
	ChipYard	PyYard
共同点	开源SoC开发设计框架，包含若干处理器核、加速器以及开发工具流等	
	基础设施：以FIRRTL作为中间层	
不同点	以Chisel（Scala）为主要开发框架	以PyChip（Python）为主要开发框架
	处理器核：Rocket、BOOM、Ariane	处理器核：PicoRV32、Ariane
	加速器：Hwacha、Gemmini、NVDLA	加速器：可编程通用加速器VTA
	验证方法：基于Scala的软件RTL仿真（Wrapper，缺乏UVM支持）	验证方法：PyHCL Tester、Cocotb（UVM支持）
	Chisel DSP	PyHCL算法设计

PyYard的思想借鉴于ChipYard的基础架构——FIRRTL的基础上，完全使用Python进行开发、测试以及验证，以达到硬件全栈开发/测试/验证的目的。

PyYard vs ChipYard



ChipYard



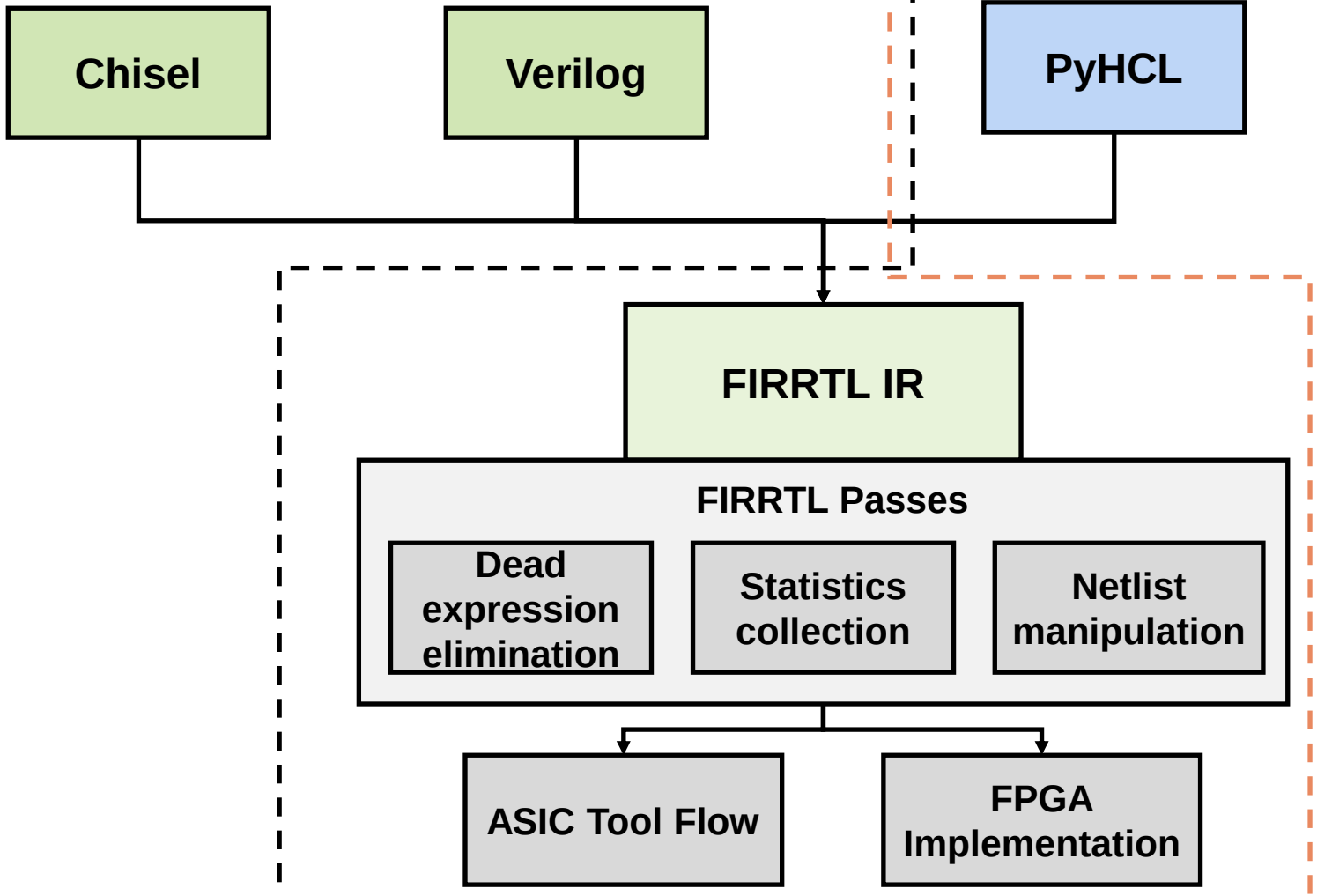
PyYard

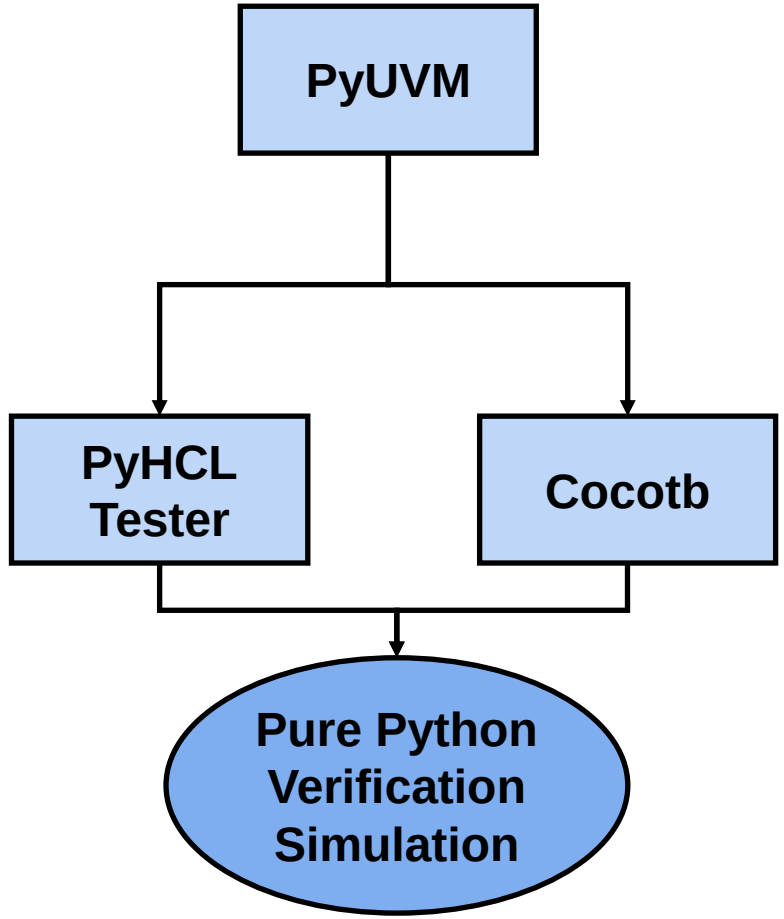
PyYard：一个具有通用可编程深度学习AI引擎（VTA）的RISC-V框架，可以支持主流的深度学习模型；本框架只需要单个通用加速器架构（VTA）可以完成大部分深度学习应用部署的需求，而不需要提供多个加速器架构使用。

PyChip vs Chisel

Chisel框架：design

PyChip框架：design & verification（Python 全栈）





目录 | CONTENT

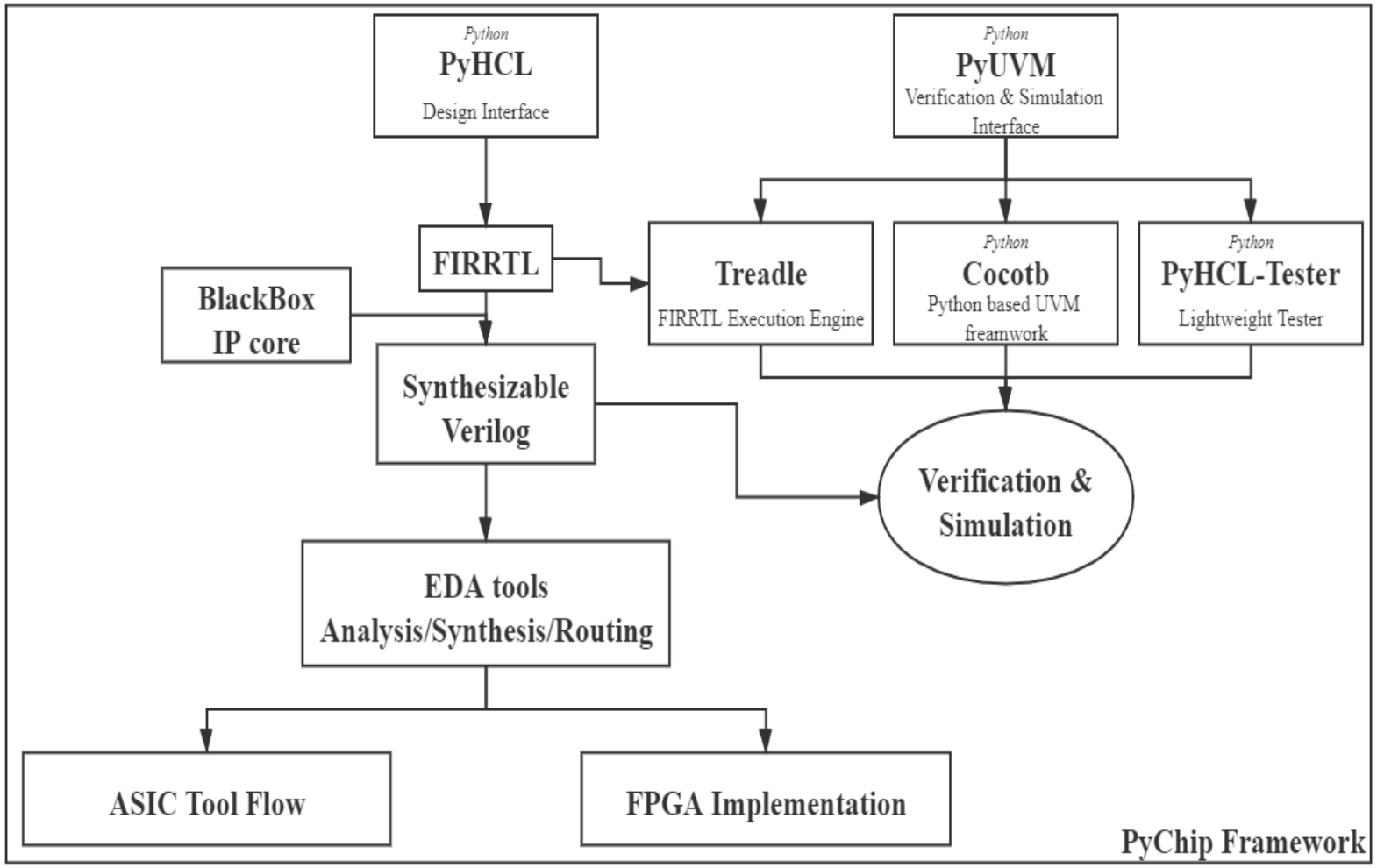
1 PyYard概述

2 PyChip框架

3 VTA体系结构

4 PyHCL与算法设计

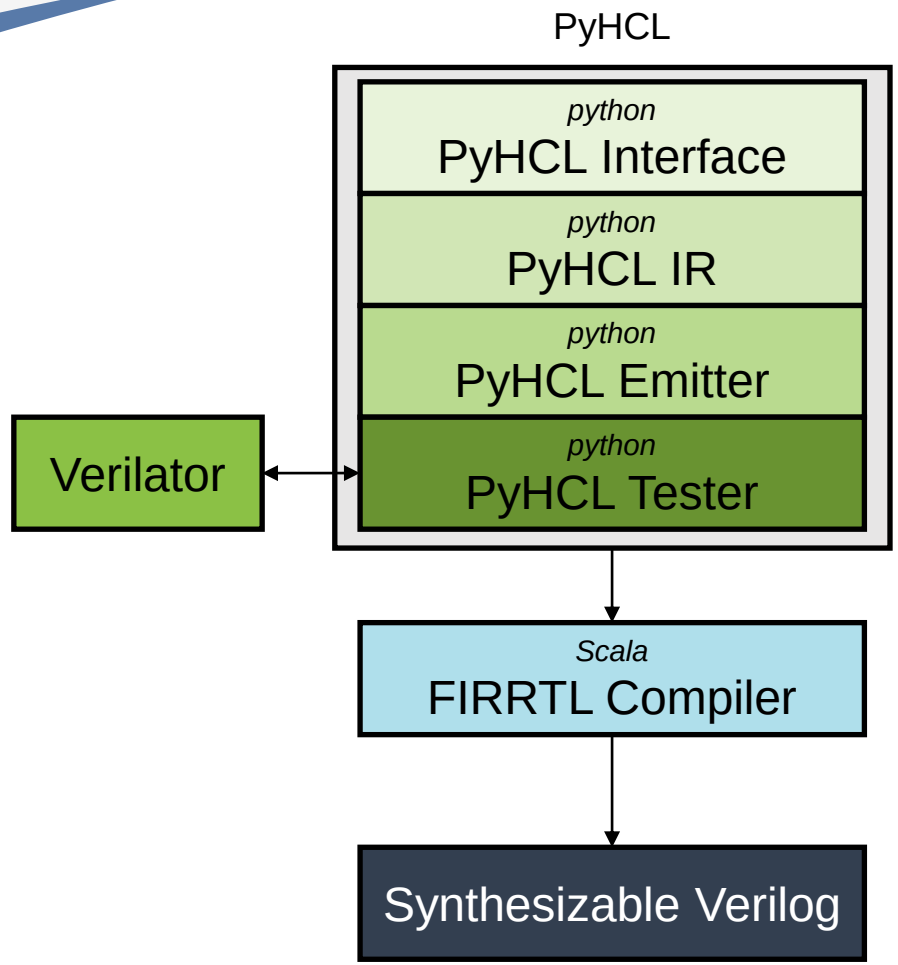
PyChip框架



PyChip整体架构

PyChip是我们团队开发一款基于Python的高层次硬件设计/仿真验证测试合一的框架，框架由两个主要的部分组成：硬件电路设计模块PyHCL以及仿真验证测试模块PyUVM。

PyChip框架

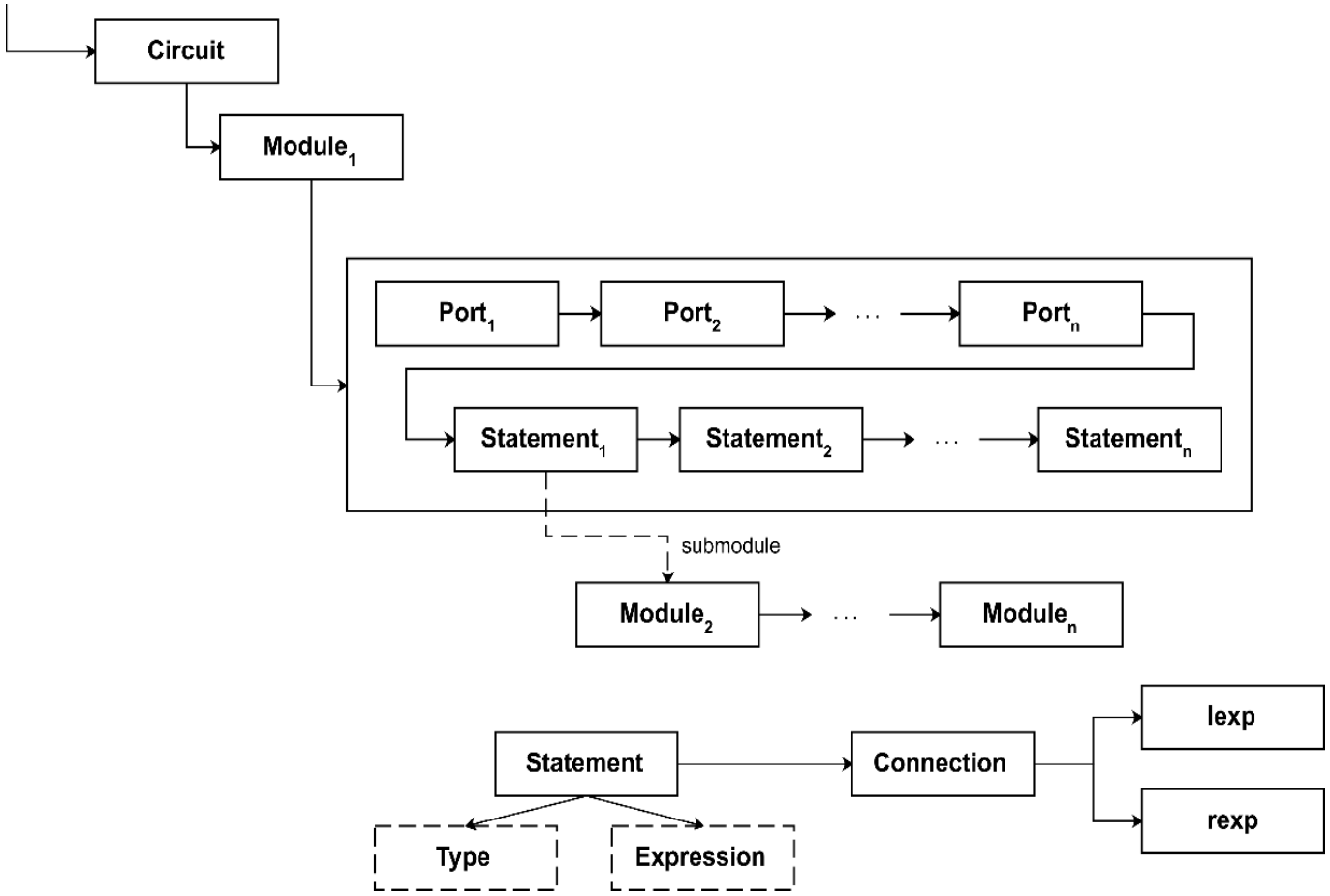


PyHCL

- PyHCL是一个使用Python设计的高层次电路设计第三方库；
- 理念与Chisel相似，都是使用构造的方式来设计电路；
- PyHCL的前端设计更为简洁精炼，且充分利用了Python语言自身的特性（动态、脚本）；
- 相较于Chisel的宿主语言Scala，PyHCL要更为轻量级。

https://github.com/scutdig/PyChip-py-hcl

PyChip框架



PyHCL IR

- PyHCL IR是PyHCL的语法树结构，用于指导内核将PyHCL代码编译为FIRRTL代码。
- PyHCL语法树实际上与FIRRTL的语素一一对应。PyHCL前端的接口都映射相应的FIRRTL语素。
- 在生成FIRRTL代码过程中，通过自顶向下的遍历语法树，便可以生成最终合法的FIRRTL代码。

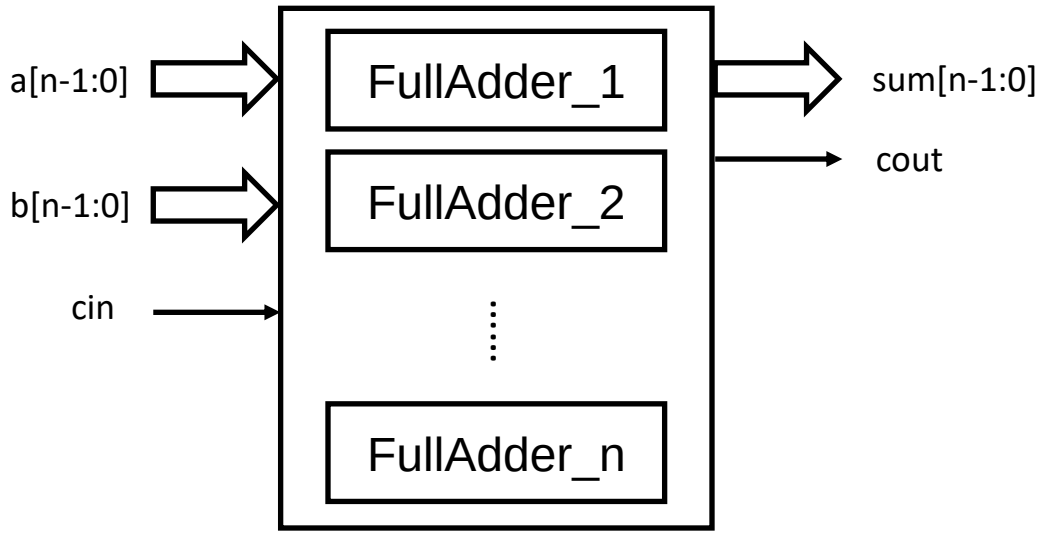
https://github.com/scutdig/PyChip-py-hcl

PyChip框架

Simple Example: Adder

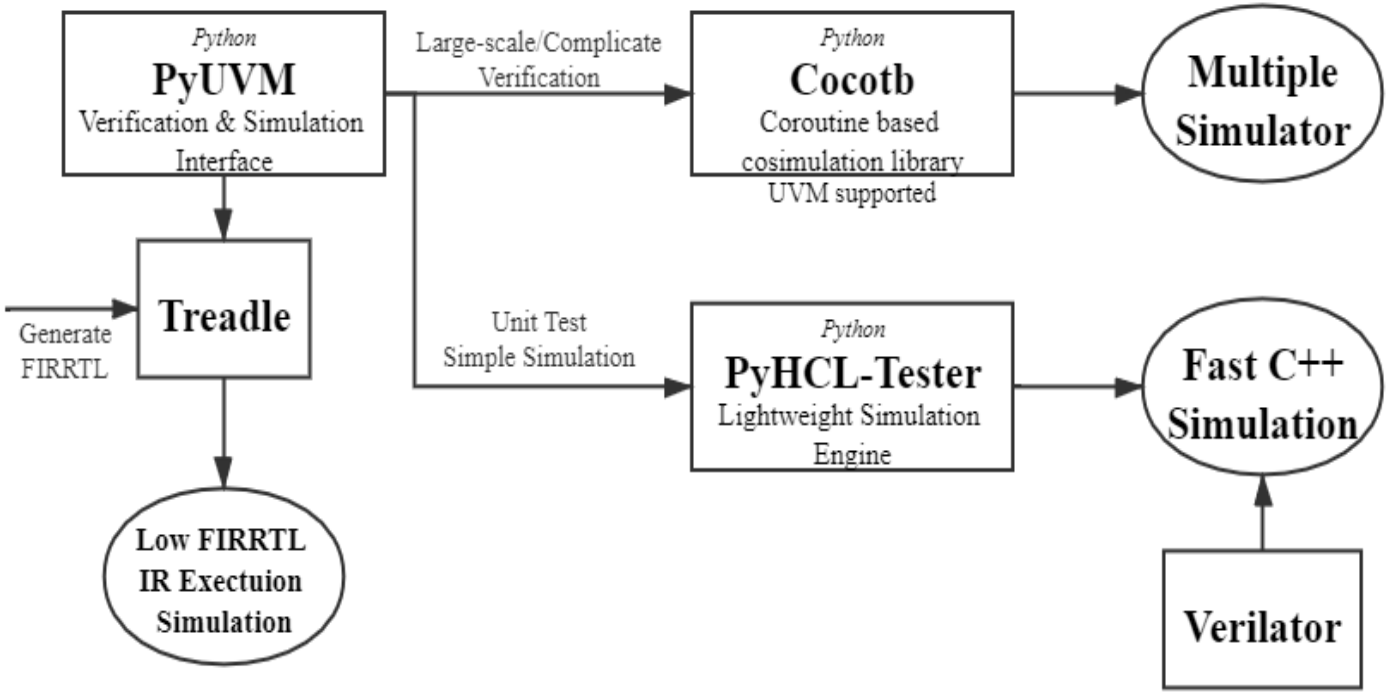
n: parameter for the width of Adder

```
class Adder(Module):
    io = IO(
        a=Input(U.w(n)),
        b=Input(U.w(n)),
        cin=Input(Bool),
        sum=Output(U.w(n)),
        cout=Output(Bool),
    )
    FAs = [FullAdder().io for _ in range(n)]
    carry = Wire(Vec(n + 1, Bool))
    sum = Wire(Vec(n, Bool))
    carry[0] <<= io.cin
    for i in range(n):
        FAs[i].a <<= io.a[i]
        FAs[i].b <<= io.b[i]
        FAs[i].cin <<= carry[i]
        carry[i + 1] <<= FAs[i].cout
        sum[i] <<= FAs[i].sum
    io.sum <<= CatVecH2L(sum)
    io.cout <<= carry[n]
```



https://github.com/scutdig/PyChip-py-hcl

PyChip框架



PyUVM

PyUVM是PyChip的仿真、验证以及测试的模块，它包含有两个主要的部件：

- 基于PyHCL的PyHCL-tester模块；
- 基于Cocotb的验证测试平台。

同时，我们集成了Treadle作为FIRRTL代码的仿真手段。

PyChip框架

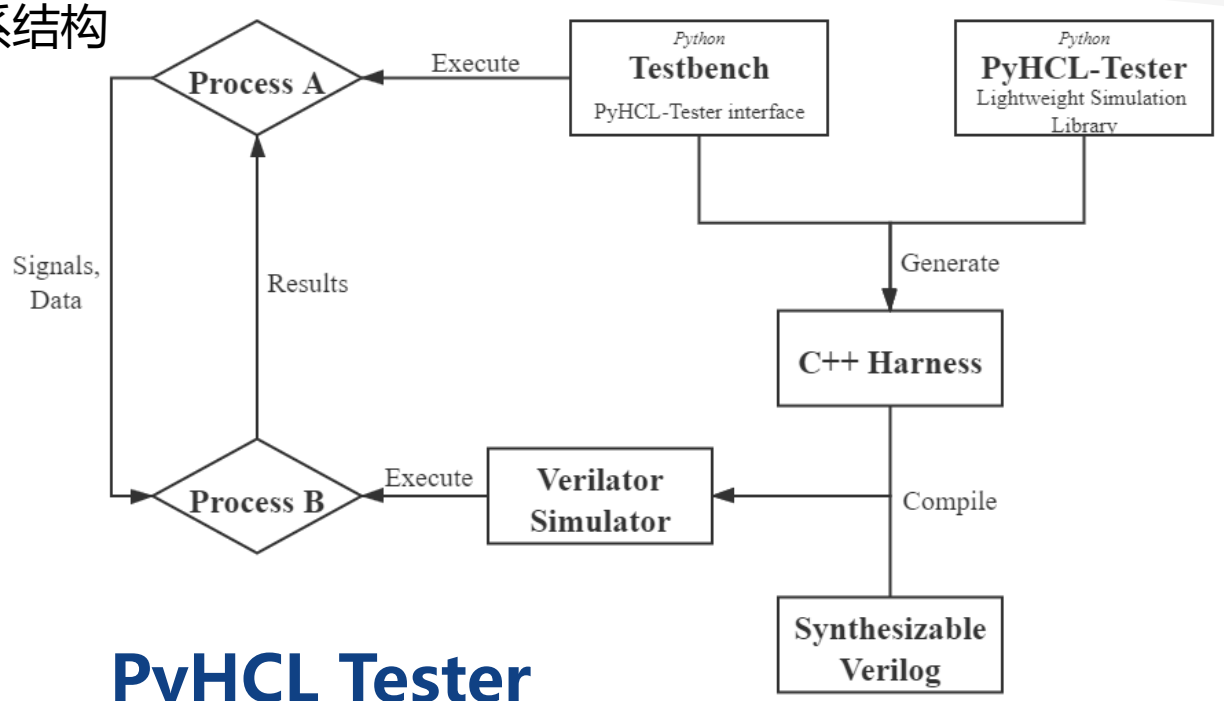
```python
def main():
    s = Simulator(adder(8))
    handler = s.handler

    # ---------- Simulation begin ---------- #
    s.start()

    for i in range(16):
        s.poke(handler.io.a, i)
        s.poke(handler.io.b, i + 1)
        s.poke(handler.io.cin, i % 2)
        s.step()
        s.peek(handler.io.sum)
        s.peek(handler.io.cout)

    s.term()
    # ---------- Simulation end ---------- #
```

示例

```
io_a-> 0
io_b-> 0
io_cin-> 0
step 0
io_sum-> 0
io_cout-> 0
---------cycle 0---------
io_a-> 0
io_b-> 0
io_cin-> 1
step 1
io_sum-> 1
io_cout-> 0
---------cycle 1---------
io_a-> 0
io_b-> 1
io_cin-> 0
step 2
io_sum-> 1
io_cout-> 0
---------cycle 2---------
io_a-> 0
io_b-> 1
io_cin-> 1
step 3
io_sum-> 0
io_cout-> 1
---------cycle 3---------
```

输出

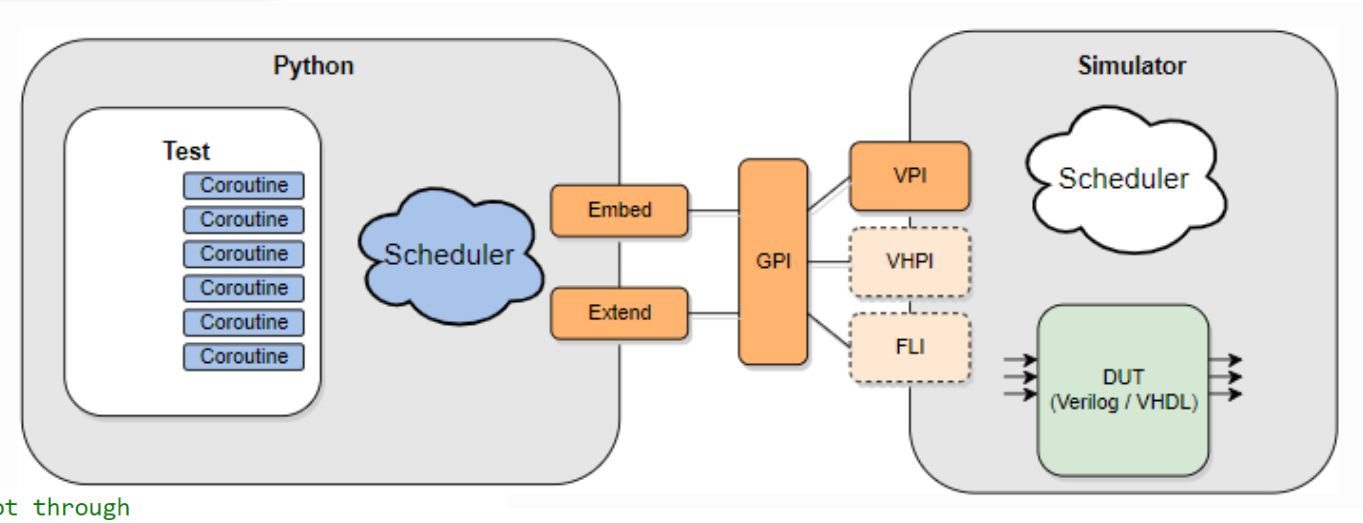
体系结构



PyHCL Tester

- PyHCL-Tester是实现在PyHCL库中的轻量级仿真测试引擎；
- 仿真引擎利用Verilator；
- 通过进程间通信的方式达到在Python接口层进行C++仿真。

PyChip框架




```
# Write to address 0 and verify that value got through
@cocotb.test(skip = False)
def write_address_0(dut):
    # Reset
    dut.rst <=  1
    dut.test_id <= 0
    axim = AXI4LiteMaster(dut, "AXIML", dut.clk)
    setup_dut(dut)
    yield Timer(CLK_PERIOD_NS * 10, units='ns')
    dut.rst <= 0
    ADDRESS = 0x00
    DATA = 0xAB
    yield axim.write(ADDRESS, DATA)
    yield Timer(CLK_PERIOD_NS * 10, units='ns')
    value = dut.dut.r_temp_0
    if value != DATA:
        # Fail
        raise TestFailure("Register at address 0x%08X should have been: \
                           0x%08X but was 0x%08X" % (ADDRESS, DATA, int(value)))
    dut.log.info("Write 0x%08X to address 0x%08X" % (int(value), ADDRESS))
```

Cocotb

- Cocotb是基于Python协程（coroutine）的对Verilog/VHDL进行协同仿真的验证测试环境。
- Cocotb是完全免费、开源的框架，且支持用户使用UVM的验证方法学：可重用以及随机生成来进行验证与测试。与UVM不同的是Cocotb框架使用Python开发的，而不是SystemVerilog。

https://github.com/cocotb/cocotb

PyChip框架

实现	Slice LUT	FF	LOC	行数占比 (PyHCL/Verilog)	工作量投入（人月）
RV32I Demo	724	1587	1149	0.366	1
PicoRV32	877	645	1383	0.454	3
LeNet-5 Demo	24918	24158	623	0.527	0.5
AXI-Full Master	56	98	302	0.467	0.2
AXI-Full Slave	44	170	201	0.203	0.2
AXI-Lite Master	54	149	215	0.346	0.2
AXI-Lite Slave	591	1192	120	0.338	0.2

使用PyHCL已经完成实现的结果展示，包括综合相关的结果、LOC（line of code）以及工作量（人力投入）等相关实现结果的展示。

https://github.com/Groupsun/PicoRV32_PyHCL

目录 | CONTENT

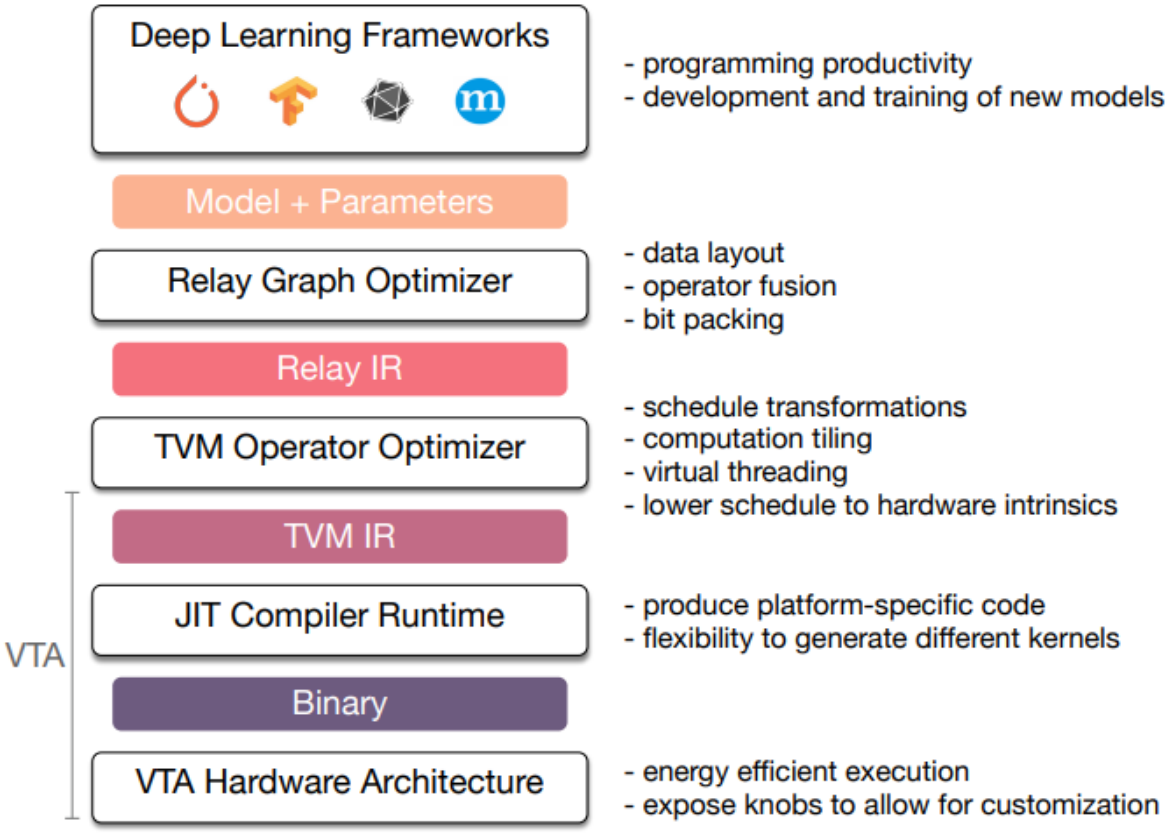
1 PyYard概述

2 PyChip框架

3 VTA体系结构

4 PyHCL与算法设计

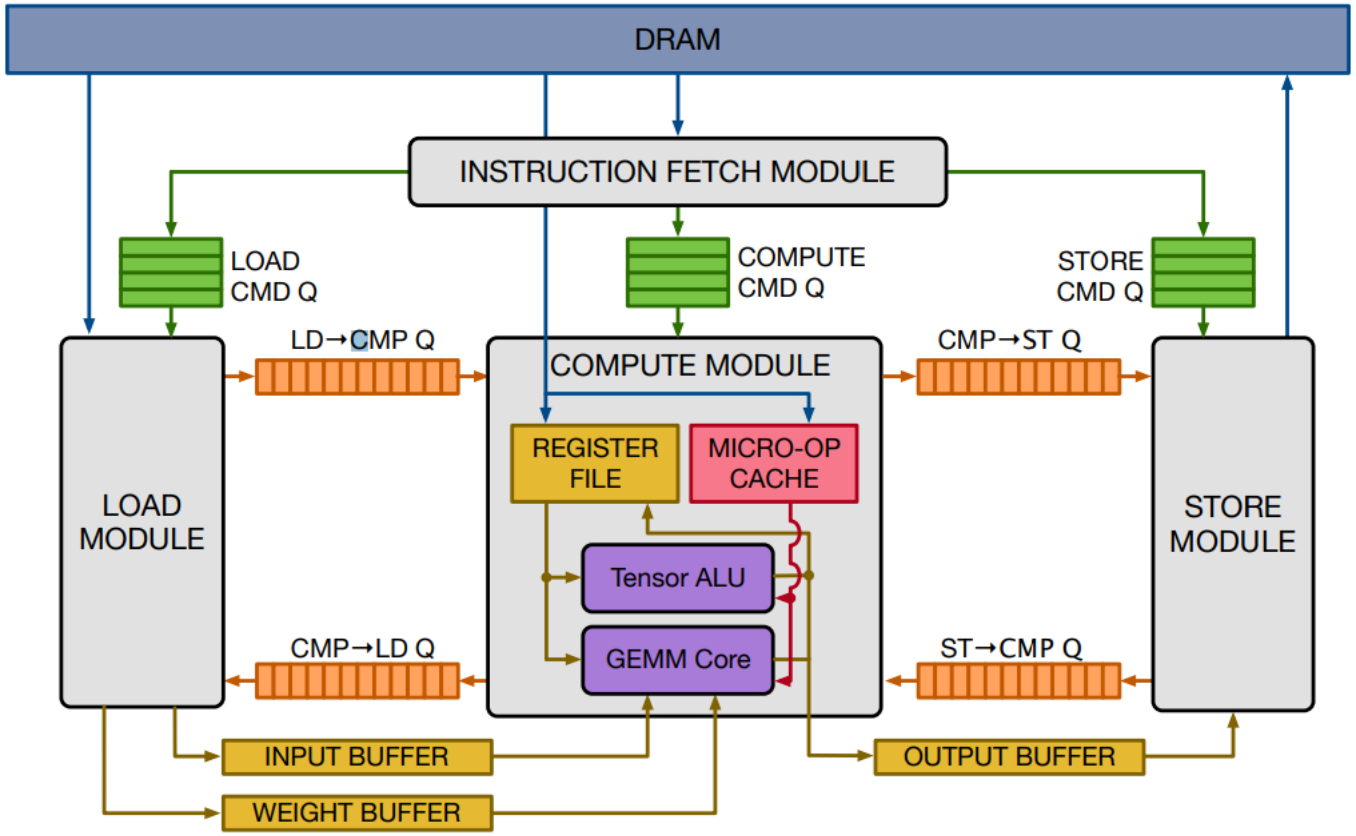
VTA体系结构



VTA+TVM：软硬件协同设计

- VTA（Versatile Tensor Accelerator）是TVM（Tensor Virtual Machine）架构的硬件扩展之一，可以借助TVM的上层工具链，构成端到端的深度学习加速器解决方案。
- “VTA+TVM”工具链支持绝大多数深度学习框架模型（PyTorch、Tensorflow、caffe、Keras、MXNet等）
- VTA工具链包括：包括TVM IR、JIT Compiler Runtime以及VTA底层硬件架构；
- TVM工具链包括：顶层模型及参数、Relay Graph Optimizer、Relay IR、TVM Operator Optimizer

VTA体系结构

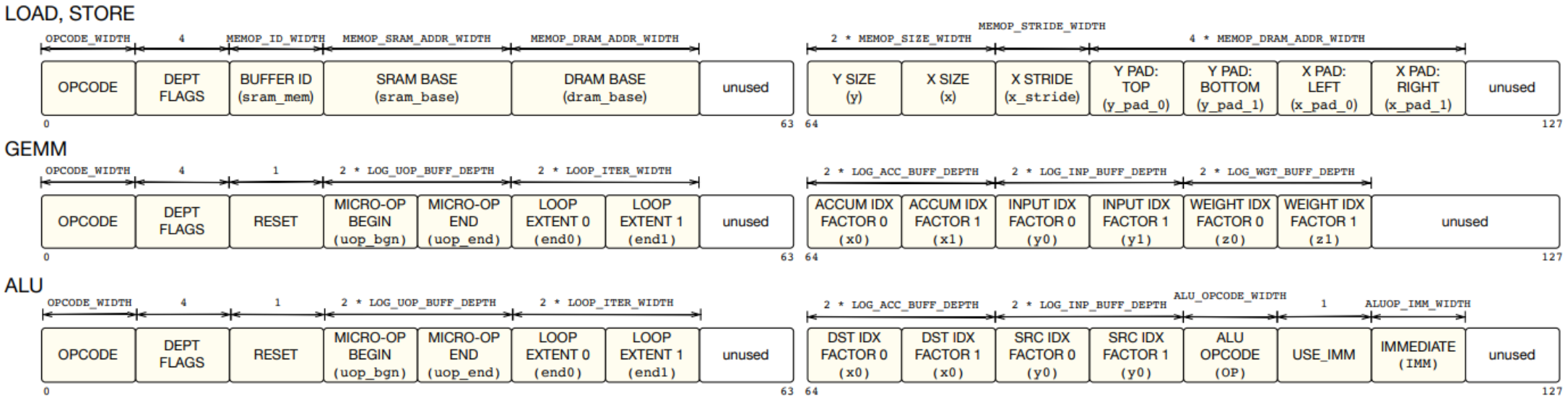


VTA体系结构设计：精简硬件

- 可编程加速器架构；
- 由四个主要模块构成：Fetch、Load、Compute以及Store。模块之间通过指令队列进行交互，并解决数据冒险问题（RAW、WAR）。
- 3级体系结构（load-compute-store）
- TPU数据流风格

https://github.com/Groupsun/vta_pyhcl

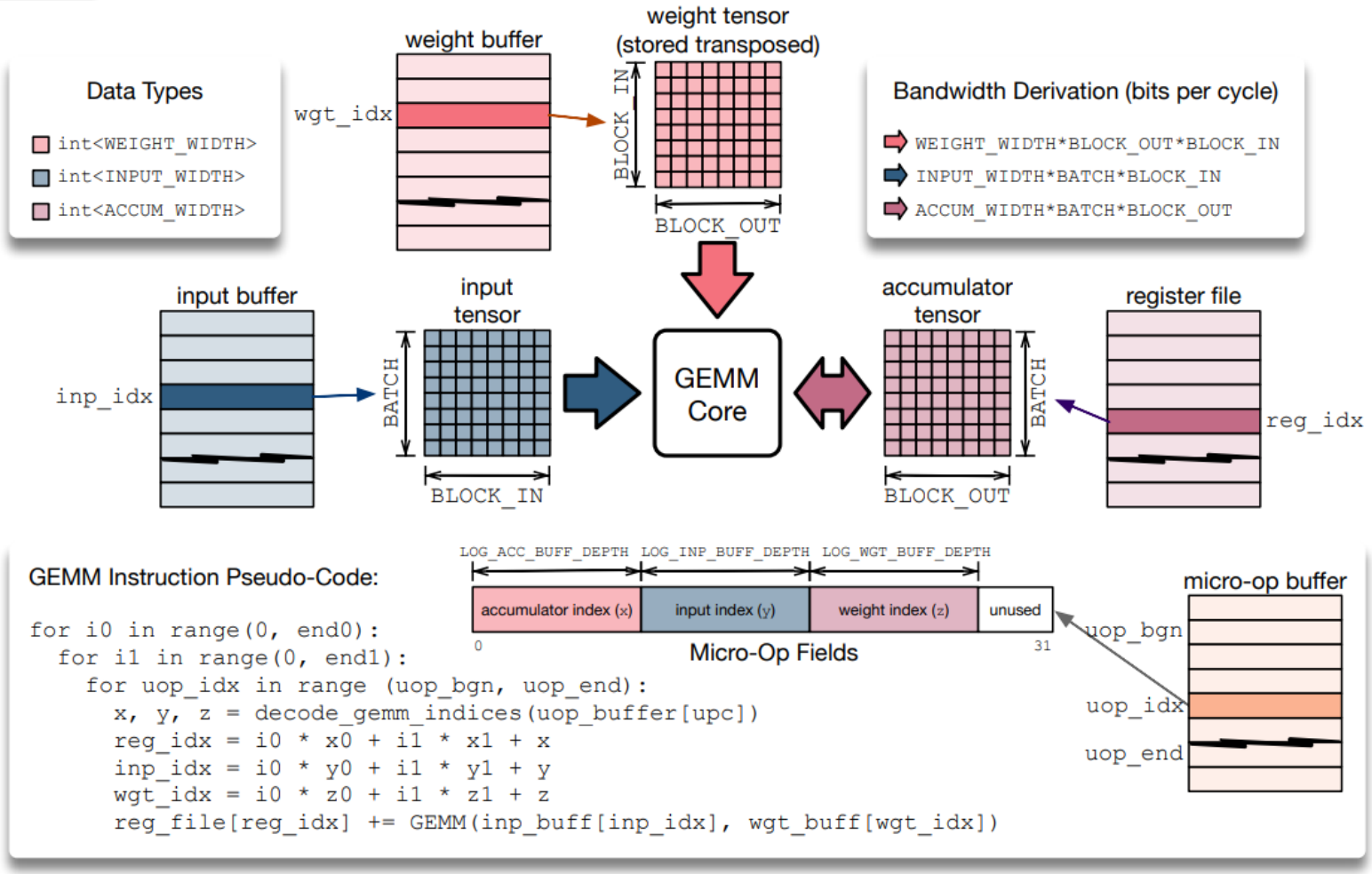
VTA体系结构



VTA ISA

- RISC微指令+CISC访存+单元张量指令混合风格；
- ISA包括4条CISC指令，其中GEMM以及ALU指令实现为类RISC微指令流的方式来进行矩阵乘法以及ALU运算等；

GEMM Core



GEMM Core用于实现神经网络中最常见的密集矩阵乘运算，并将结果写回。寻址通过一系列的微指令完成，用于适配不同的神经网络模型以及运算操作。

https://github.com/Groupsun/vta_pyhcl

目录 | CONTENT

1 PyYard概述

2 PyChip框架

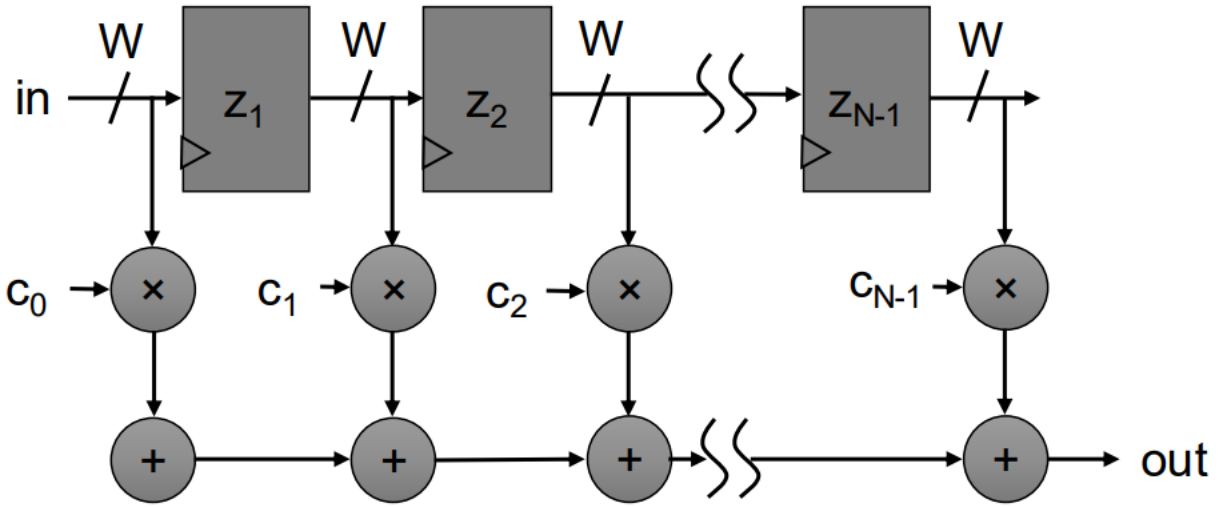
3 VTA体系结构

4 PyHCL与算法设计

PyHCL与算法设计

```
def myManyDynamicElementVecFir(length: int):
    class MyManyDynamicElementVecFir(Module):
        io = IO(
            i=Input(U.w(8)),
            valid=Input(Bool),
            o=Output(U.w(8)),
            consts=Input(Vec(length, U.w(8))),
        )

        taps = [io.i] + [RegInit(U.w(8)(0)) for _ in range(len(io.consts) - 1)]
        for a, b in zip(taps, taps[1:]):
            with when(io.valid):
                b <<= a

        m = map(lambda x: x[0] * x[1], zip(taps, io.consts))
        io.o <<= reduce(lambda x, y: x + y, m)

    return MyManyDynamicElementVecFir()
```

Example：参数化FIR滤波器



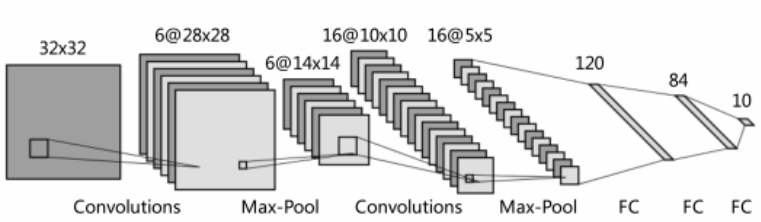
Python原生的动态性以及函数式编程，给予PyHCL以逼近高级语言的方式描述硬件算法核的设计。

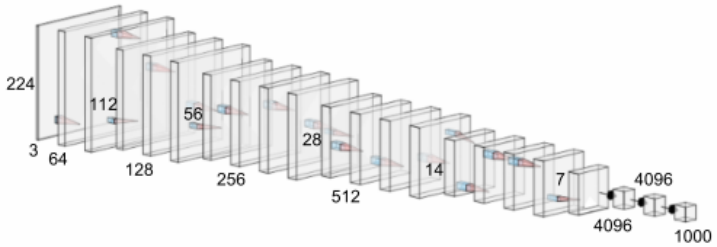
我们使用PyHCL对VTA进行重构，探索PyHCL在TPU设计描述方面的可能性。

PyHCL与算法设计

探究使用PyHCL描述不同神经网络当中的基本单元，配置为参数化的生成器（Generator），达到快速重构神经网络的作用。

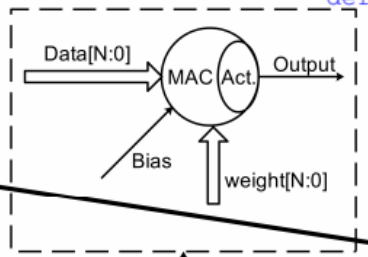
图左侧为使用PyHCL描述的LeNet-5神经网络结构，右侧为VGG-16神经网络结构，两者之间可以通过参数化生成器在顶层模块快速配置实现不同神经网络结构之间的转换。



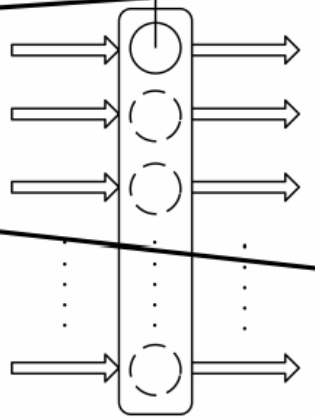


Top-level network code change

```
def lenet_top():
    class Top(Module):
        io = IO(
            # IO ports declarations
        )
        conv1 = Conv(width=16, shape=(5, 5, 1,
            6), strides=1)
        act1 = Tanh(width=16)
        max_pool1 = MaxPool(width=16, psize=2,
            strides=2)
        # Connections of convolutional layer1
        # ... ...
        conv2 = Conv(width=16, shape=(5, 5, 6,
            16), strides=1)
        act2 = Tanh(width=16)
        max_pool2 = MaxPool(width=16, psize=2,
            strides=2)
        # Connections of convolutional layer2
        # ... ...
        flatten = Flatten(width=16, shape=(5,
            5, 16))
        fc1 = FC(width=16, input_num=400,
            output_num=120)
        act3 = Tanh(width=16)
        fc2 = FC(width=16, input_num=120,
            output_num=84)
        # fc3 ...
        # Connections of full connection layers
        # ... ...
    return Top()
```

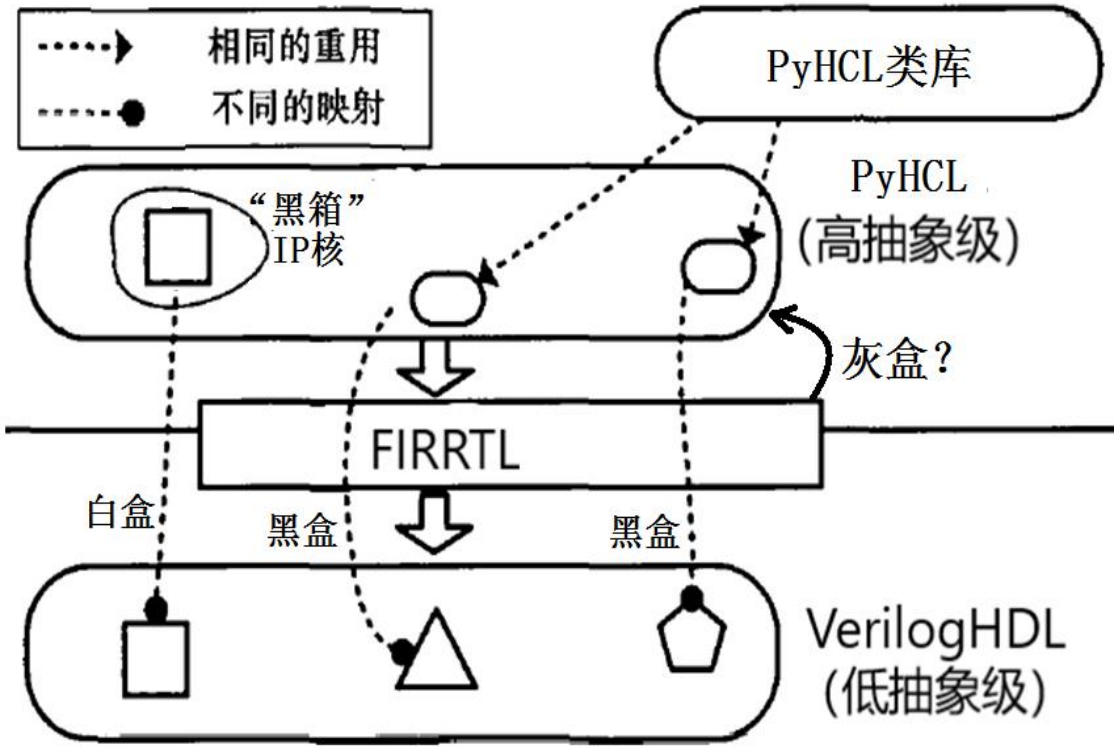


Layer



```
def vgg16_top():
    class Top(Module):
        io = IO(
            # IO ports declarations
        )
        # ConvProc and ConvLayer constructed by
            basic computing components (Conv,
            Tanh)
        conv_process_1 = ConvProc(width=16,
            conv_count=2, shape=(3, 3, 3, 64),
            strides=(1, 2), act=1, psize=2)
        conv_process_2 = ConvProc(width=16,
            conv_count=2, shape=(3, 3, 64,
            128), strides=(1, 2), act=1,
            psize=2)
        # Conv Process 3, 4, 5 ...
        fc_conv_6 = ConvLayer(width=16,
            shape=(7, 7, 512, 4096), strides=1,
            act=1)
        fc_conv_7 = ConvLayer(width=16,
            shape=(1, 1, 4096, 4096),
            strides=1, act=1)
        fc_conv_8 = ConvLayer(width=16,
            shape=(1, 1, 4096, 1000),
            strides=1, act=1)
        softmax = SoftMax(width=16)
        # Connections
        # ...
    return Top()
```

未来工作的展望



灰盒(PyHCL) vs 黑盒(HLS) vs 白盒(Verilog)

- 设计PyHCL的用户接口层，生成相应的FIRRTL代码，并利用Diagrammer以及Treadle设计可视化的调试工具并反馈到设计接口层（灰盒设计）。
- 黑盒设计模式（HLS）：根据高级语言算法代码分析产生对应的算法硬件代码，生成的内核逻辑未知（黑盒）。
- 白盒设计模式（Verilog）：直接对硬件逻辑建模进行设计，底层逻辑完全暴露（白盒）。
- 灰盒设计模式(PyHCL)：在“黑盒”设计基础上，在FIRRTL级提供一个可视化和动态调试反馈，通过FIRRTL数据流可视化和序列化动态仿真，实现对底层硬件数据流动的“透视”，方便设计者调整高抽象级的PyHCL代码。

谢谢！