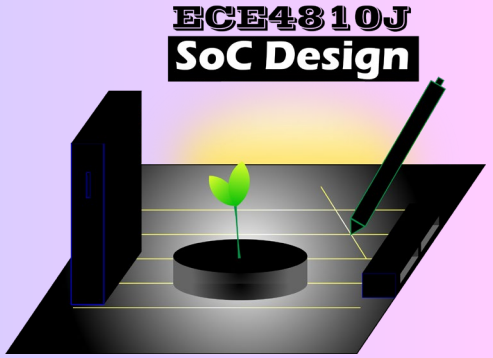
Lab #5
ASIC Backend Flow

ECE4810J System-on-Chip Design



Yihua Liu
UM-SJTU Joint Institute
ayka_tsuzuki@sjtu.edu.cn
Oct. 31, 2022

Overview

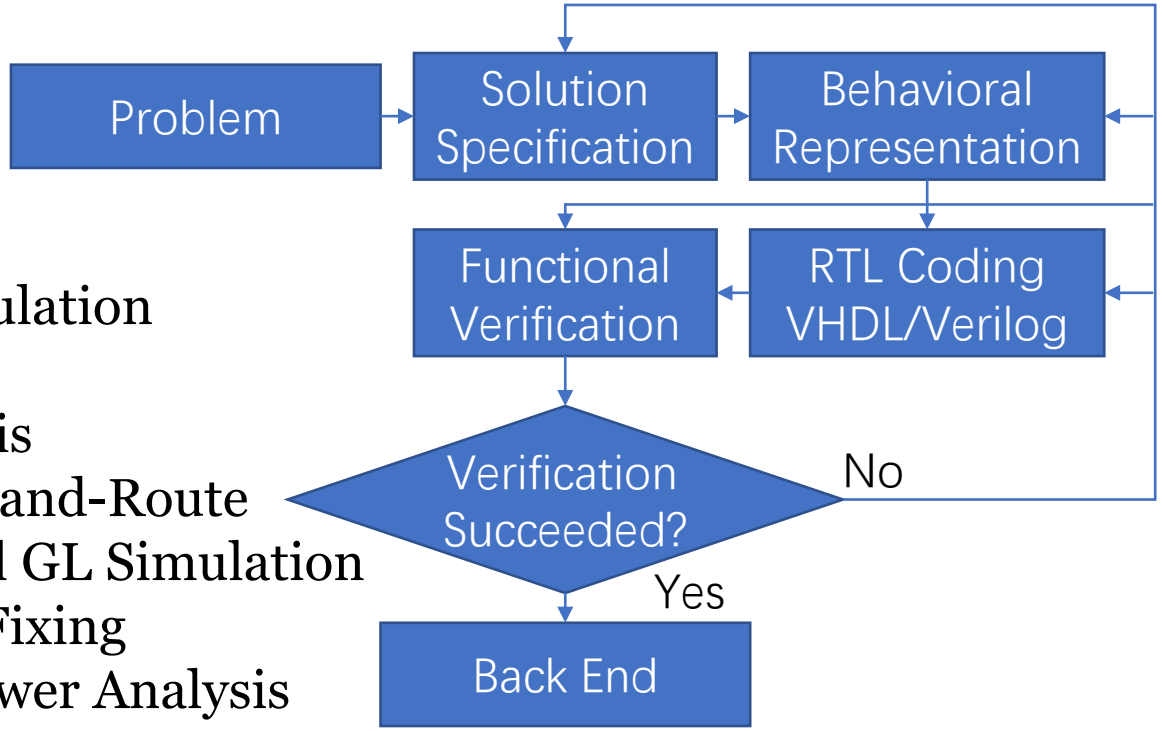
ASIC Flow Front-End vs Back-End

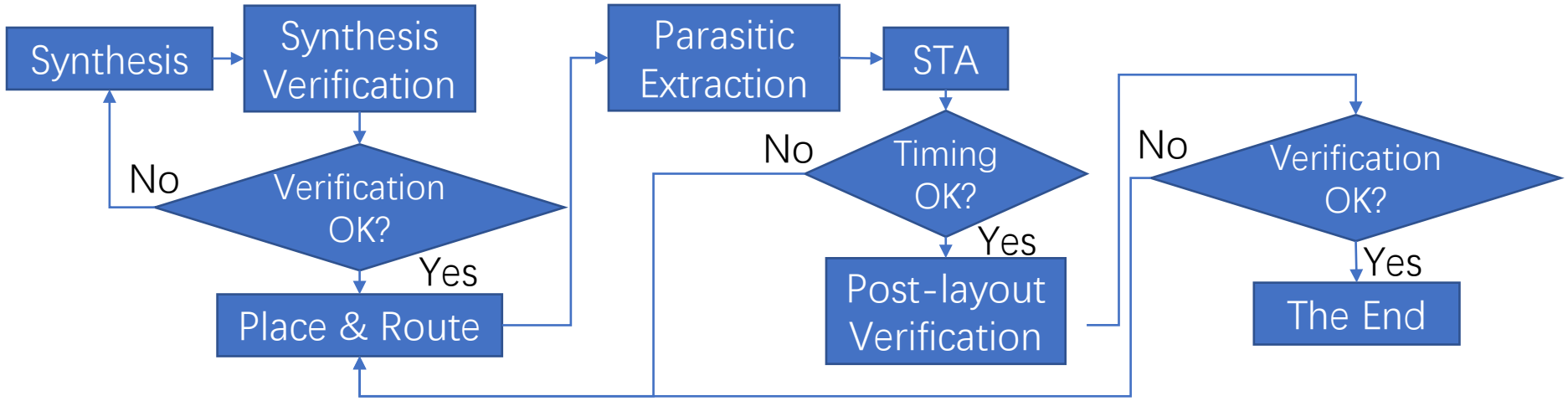
Front-End:

- Hardware modeling in HDL
- Synopsys VCS for 4-State RTL Simulation
- Synopsys Design Compiler for Synthesis
- Synopsys VCS for Fast-Functional Gate-Level Simulation

Back-End:

- Synopsys PrimeTime for Post-Synth Power Analysis
- Synopsys IC Compiler/Cadence Innovus for Place-and-Route
- Synopsys VCS for Back-Annotated Fast-Functional GL Simulation
- Synopsys PrimeTime for STA & Performing Hold Fixing
- Synopsys PrimeTime for Post-Place-and-Route Power Analysis



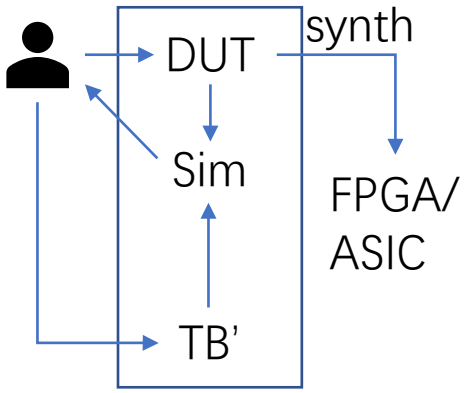


VLSI Design Methodologies

JOINT INSTITUTE 交大密西根学院

HDL
Hardware Description Language

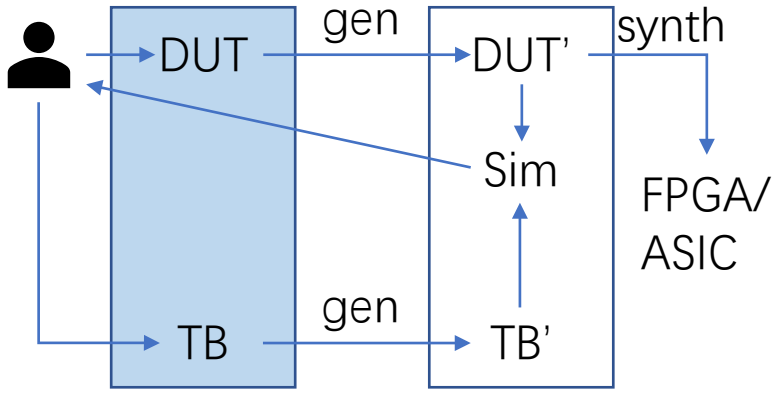
HDL (Verilog)



- ✓ Fast edit-sim-debug loop
- ✓ Single language for structural, behavioral, + TB
- ✗ Difficult to create highly parameterized generators

HPF
Hardware Preprocessing Framework

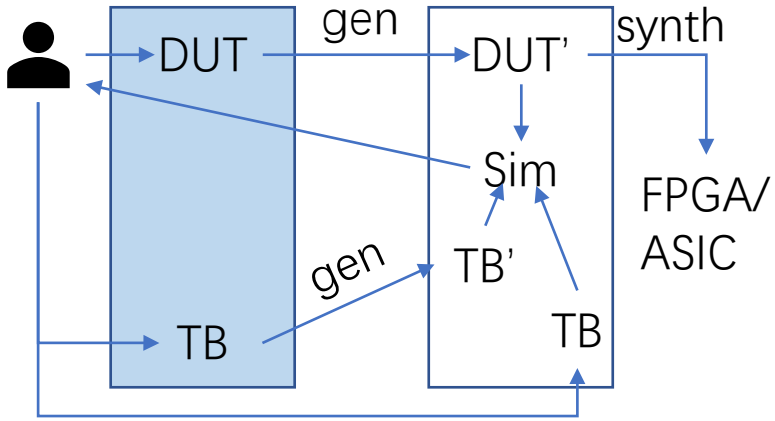
Mixed (Verilog+Perl) → HDL (Verilog)



- ✗ Slower edit-sim-debug loop
- ✗ Multiple languages create "semantic gap"
- ✓ Easier to create highly parameterized generators

HGF
Hardware Generation Framework

Host Language (Scala) → HDL (Verilog)



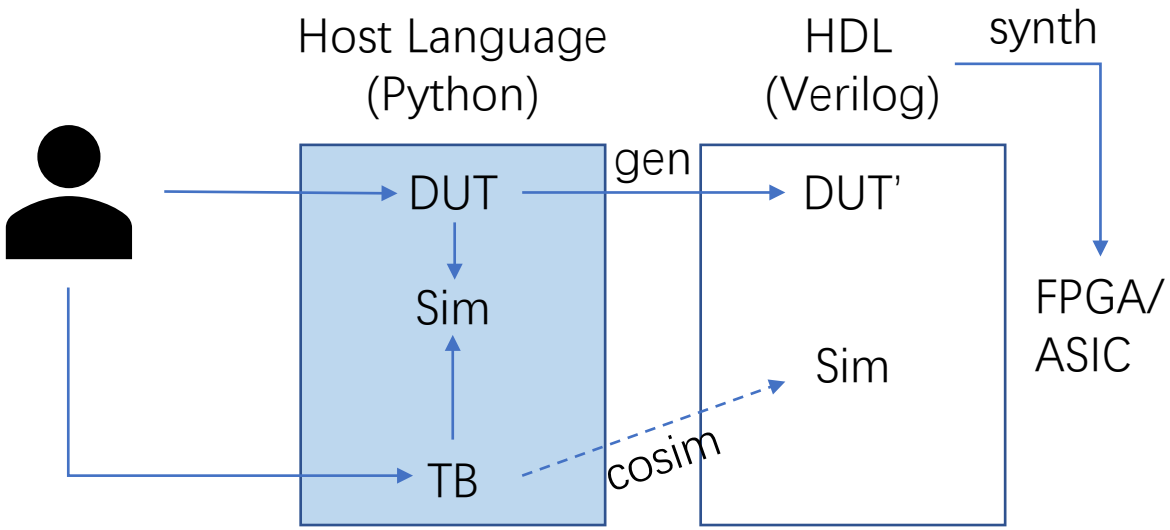
- ✗ Slower edit-sim-debug loop
- ✓ Single language for structural, behavioral, + TB
- ✓ Easier to create highly parameterized generators
- ✗ Cannot use power of host language for verification

Productive MLM & VLSI Design

Multi-Level Modeling
SystemC

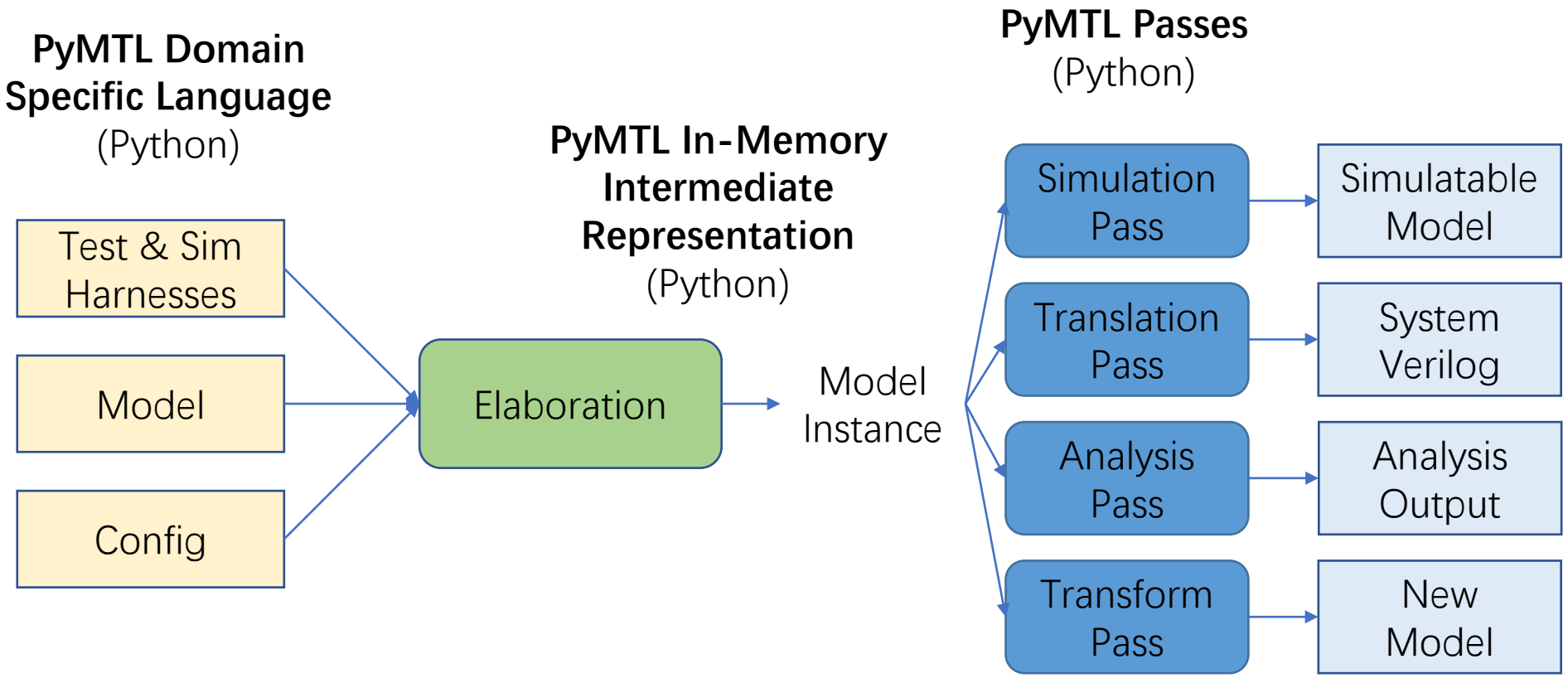
VLSI Design
Chisel

HGSF
Hardware Generation and Simulation Framework

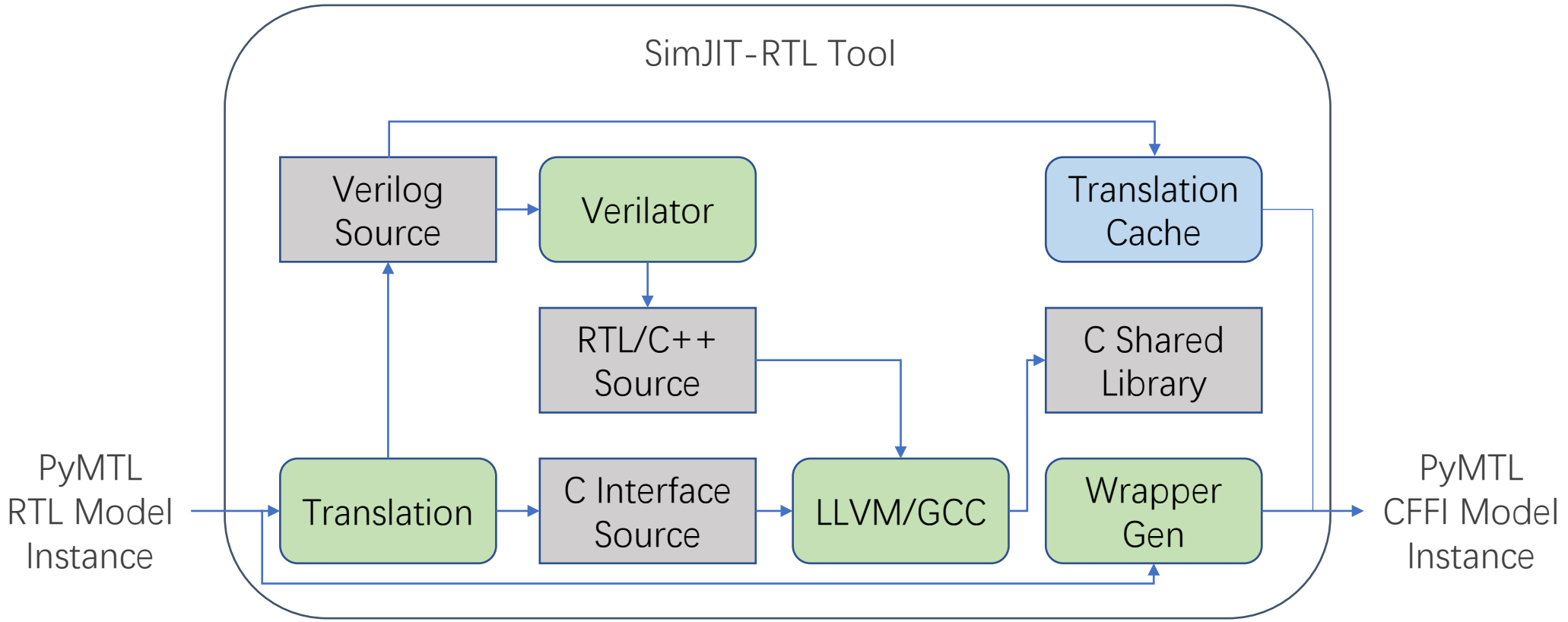


- Single framework for ML modeling & VLSI design
- Fast edit-sim-debug loop
- Single language for structural, behavioral, + TB
- Easy to create highly parameterized generators
- Use power of host language for verification

PyMTL3 Framework



PyMTL/Verilog Integration



PyMTL3: Levels of Abstraction

PyMTL3 hardware modeling framework can be used for functional-level modeling, verification, and simulator harnesses. Computer architects can model systems at various levels of abstraction including at the: functional-level (FL), cycle-level (CL), and register-transfer-level (RTL).

Functional-Level – FL models implement the *functionality* but not the timing of the hardware target. FL models are useful for exploring algorithms, performing fast emulation of hardware targets, and creating golden models for verification of CL and RTL models. FL models can also be used for building sophisticated test harnesses. FL models are usually the easiest to construct, but also the least accurate with respect to the target hardware.

Cycle-Level – CL models capture the *cycle-approximate behavior* of a hardware target. CL models will often augment the functional behavior with an additional timing model to track the performance of the hardware target in cycles. CL models are usually specifically designed to enable rapid design-space exploration of cycle-level performance across a range of microarchitectural design parameters. CL models attempt to strike a balance between accuracy, performance, and flexibility.

Register-Transfer-Level – RTL models are *cycle-accurate*, *resource-accurate*, and *bit-accurate* representations of hardware. RTL models are built for the purpose of verification and synthesis of specific hardware implementations. RTL models can be used to drive EDA toolflows for estimating area, energy, and timing. RTL models are usually the most tedious to construct, but also the most accurate with respect to the target hardware.

PyMTL3: Synthesizability

Keep in mind that PyMTL3 is embedded within Python, which is a fully general-purpose language. Given this, it is very easy to write PyMTL3 code that does not actually model any kind of realistic hardware. **Students must be very diligent in actively deciding whether or not they are writing synthesizable RTL models or non-synthesizable code. Students must always keep in mind what hardware they are modeling and how they are modeling it!**

Concurrent blocks: *model* hardware `update_ff, update, update_once`, accept non-synthesizable code only for debugging, assertions, or line tracing

Elaboration code: outside concurrent blocks, *generate* hardware, accept any code

Always allowed	Allowed with Limitations	Explicitly not Allowed
`Bits bitstruct`	accessing Pyt	`**= //=`
`& | ^ ~ + - >> <<`	writing signals	`and or not while break continue`
`== != > >= < <=`	writing temporary va	`ass try except raise`
`reduce_and() reduce_or() reduce_xor() if else elif`	reading `reset` signal	`as is in with return yield import from del exec pass lambda finally`
`sext() zext() concat()`	read-modify-write signal	constructing Python lists/dicts, using dicts
`s.signal[n] s.signa[n:m]`	`*`	reading/writing non-signals/`clk` signal
reading constvars/signals	`for`	writing `reset` signal

read-modify-write not currently supported

Must use `<<=` in `update_ff`
Must use `@=` in `update`

Use `& | ~` instead

only in (sync) `update_ff`
not in (async) `update`

Can synthesize into quite a bit of logic with statically known bounds

Instances of `InPort`, `OutPort`, or `Wire`

PyMTL3 Basics

JOINT INSTITUTE 交大密西根学院

```
% python
>>> from pymtl3 import *
```

`Bits` and `Bitstruct` Data Type

Most HDLs support four-state values (0, 1, X, Z)

PyMTL3 supports two-state values (0, 1)

`Bits1, Bits2,…, Bits255`

```
>>> a = Bits32( 0xabcd0123 )
>>> a[24:32] = 0x67
>>> b = Bits32( a )
>>> a = Bits8( 0b10101100 )
>>> reduce_and(a)
Bits1(0x0)
>>> a = Bits8( 0xab )
>>> b = Bits12( 0xcde )
>>> concat( a, b )
Bits20(0xabcde)
>>> a = Bits4( 0xa )
>>> sext( a, 8 )
Bits8(0xfa)
>>> zext( a, 8 )
Bits8(0x0a)
>>> a = Bits8( 0xff )
>>> trunc( a, 3 )
Bits3(0x7)
>>> @bitstruct
... class Point:
...   x: Bits4
...   y: Bits4
...
>>> pt1 = Point(3, 4)
>>> pt1
Point(Bits4(0x3),Bit
s4(0x4))
>>> str(pt1)
'3:4'
>>> pt1.x
Bits4(0x3)
>>> pt1.to_bits()
Bits8(0x34)
>>> Point.from_bits( Bits8(0x34) )
Point(Bits4(0x3),Bits4(0x4))
>>> nbits = 8
>>> PointN =
mk_bitstruct( f"Point{nbits}", {
...   'x': mk_bits(nbits),
...   'y': mk_bits(nbits),
... })
...
>>> pt2 = PointN( 3, 4 )
>>> pt2
Point8(Bits8(0x03),Bits8(0x04))
>>> pt2.to_bits()
Bits16(0x0304)
```

Reference

① University of Porto. “desing_flow_LuisGomes_v1.” Mar. 2022.
② Christopher Batten. “ASIC Flow Front-End.” 2022.