ECE 5745 Complex Digital ASIC Design
PyMTL3 Usage Rules

School of Electrical and Computer Engineering
Cornell University

revision: 2021-03-04-12-30

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. Indeed, we actually need this feature to be able to write clean and productive functional-level models, test harnesses, assertions, and line tracing. **So students must be very diligent in actively deciding whether or not they are writing synthesizable register-transfer-level models or non-synthesizable code. Students must always keep in mind what hardware they are modeling and how they are modeling it!**

Students' design work will almost exclusively use synthesizable PyMTL3 register-transfer-level (RTL) models. Note that students can use any Python code they like in their elaboration code; the elaboration code is all of the Python code *outside* the PyMTL3 concurrent blocks (i.e., outside `update_ff` and `update` blocks). This is because elaboration code is used to *generate* hardware instead of actually *model* hardware. It is also acceptable to include a limited amount of non-synthesizable code in concurrent blocks for the sole purpose of debugging, assertions, or line tracing. If the student includes non-synthesizable code in their concurrent blocks, they should demarcate this code with comments. This explicitly documents the code as non-synthesizable and aids automated tools in removing this code before synthesizing the design. **If at any time students are unclear about whether a specific construct is allowed in a synthesizable concurrent block, they should ask the instructors.**

The next page includes a table that outlines which Python constructs are allowed in synthesizable PyMTL3 concurrent blocks, which constructs are allowed in synthesizable PyMTL3 concurrent blocks with limitations, and which constructs are explicitly not allowed in synthesizable PyMTL3 concurrent blocks.

Unlike ECE 4750, these rules are more of a suggestion than hard rules. Students are allowed to use anything that PyMTL3 can translate into Verilog and that Synopsys Design Compiler can synthesize. If you figure out that Synopsys Design Compiler can synthesize a more sophisticated syntax that significantly simplifies your design, then by all means use that syntax.

Always Allowed in Synthesizable Concurrent Blocks	**Allowed in Synthesizable Concurrent Blocks With Limitations**	**Explicitly Not Allowed in Synthesizable Concurrent Blocks**
`Bits`	accessing Python lists[2]	`/ // % **`
`bitstruct`	writing signals[3]	`+= -= *= /= %= **= //=`[9]
`& | ^ ~`	writing temporary variables[4]	`and, or, not`[10]
`+ -`	reading `reset` signal[5]	`while, break, continue`
`>> <<`	read-modify-write `signal`[6]	`def, global, class`
`== != > <= < <=`	`*`[7]	`try, except, raise`
`reduce_and(), reduce_or()`	`for`[8]	`as, is, in`
`reduce_xor()`		`with, return, yield`
`sext(), zext(), concat()`		`import, from`
`if, else, elif`		`del, exec, pass`
`s.signal[n], s.signal[n:m]`		`lambda`
reading constant variables		`finally`
reading signals[1]		constructing Python lists
		constructing/using Python dicts
		reading/writing non-signals[11]
		reading/writing `clk` signal
		writing `reset` signal

1 Signals are instances of `InPort`, `OutPort`, or `Wire`. PyMTL3 interfaces group ports together, so accessing members of an interface is allowed. Signals can only communicate bit-specific value types (e.g., `Bits`, `bistruct`).

2 Accessing lists of signals or lists of models is allowed although students should be careful to keep the indexing logic relatively simple.

3 Signals must be written using the `<<=` operator in a `update_ff` concurrent block, and must be written using the `@=` operator in a `update` concurrent block.

4 Writing temporary variables is allowed as long as the type of the temporary variable (e.g., the bitwidth) can be reasonably inferred.

5 Reading the special `reset` signal is allowed, but only in a `update_ff` concurrent block. Reading the `reset` signal in a `update` concurrent block is not allowed. If you need to factor the reset signal into some combinational logic, you should instead use the `reset` signal to reset some state bit, and the output of this state bit can be factored into some combinational logic. In other words, students should only use synchronous and not asynchronous resets.

6 Reading a signal, performing some arithmetic on the corresponding value, and then writing this value back to the same signal (i.e., read-modify-write) is not allowed within an `update` concurrent block. This is a combinational loop and does not model valid hardware. Read-modify-write is allowed in an `update_ff` concurrent block using `<<=`, although we urge students to consider separating the sequential and combinational logic. Students can use an `update` concurrent block to read the signal, perform some arithmetic on the corresponding value, and then write a temporary wire; and use an `update_ff` concurrent block to flop the temporary wire into the destination signal.

7 Be careful using the `*` operator since it can synthesize into quite a bit of logic.

8 `for` loops with statically known bounds may be synthesizable, although students should use great care and clearly understand what hardware they are modeling.

9 These assignment operators essentially perform a read-modify-write of a signal. See the above footnote. Technically, these operators might model valid hardware if used within a `update_ff`, but this syntax is not currently supported and will result in strange simulator behavior. Therefore, these assignment operators are never allowed in synthesizable concurrent blocks.

10 Use the `&`, `|`, `~` operators instead of the `and`, `or`, `not` operators.

11 Students cannot use non-signals (i.e., normal Python variables) to communicate between concurrent blocks. Students must use instances of `InPort`, `OutPort`, `Wire`. PyMTL3 interfaces group ports together, so accessing members of an interface is allowed.