PyMTL3

Release 0.5.8

May 19, 2020

First Steps

PyMTL3 is the latest version of PyMTL, an open-source Python-based hardware generation, simulation, and verification framework with multi-level hardware modeling support.

CHAPTER 1

First Steps

Installation and quick start in a Python REPL.

- **Installing PyMTL3 and Verilator**: *Installation guide*
- **Using PyMTL3 in a Python REPL**: *Quick start*

1.1 Installation

1.1.1 Install the PyMTL3 Framework

PyMTL3 requires >= Python3.6. We highly recommend working inside a virtual environment. Here is how to create a virtual environment and install PyMTL3 using `pip`:

```
$ python3 -m venv pymtl3
$ source pymtl3/bin/activate
$ pip install pymtl3
```

Note: Your virtual environment will be deactivated when your current terminal session is termindated. To re-activate your PyMTL3 virtual environment:

```
$ source pymtl3/bin/activate
```

and to manually deactivate your current virtual environment:

```
$ deactivate
```

1.1.2 Install the Verilator Simulator

PyMTL3 optionally supports Verilator, an open-source toolchain for compiling Verilog RTL models into C++ simulators, to co-simulate Verilog modules with PyMTL3 components. We recommend compiling and installing the latest

Verilator from source because the standard package managers tend to host only the old versions which lack some of the features that PyMTL3 uses. Here is how to compiling the latest Verilator and setting it up to work with PyMTL3:

Acquire the necessary packages

We will be using `git` to obtain the latest release of Verilator. In addition, PyMTL3 uses cffi to communicate with the C++ simulator generated by Verilator.

```
$ sudo apt-get install git python-dev libffi-dev
```

We also need the following packages to compile Verilator from source:

```
$ sudo apt-get install make autoconf g++ libfl-dev bison
```

Build and install Verilator

First retrieve the most recent stable Verilator release from the official git repo:

```
$ git clone https://git.veripool.org/git/verilator
$ git pull
$ git checkout stable
```

Next build and install Verilator:

```
$ autoconf
$ ./configure
$ make
$ sudo make install
```

Verify that Verilator is on your `PATH` as follows:

```
$ verilator --version
```

Set up Verilator for PyMTL3

There are two ways to set up Verilator for PyMTL3: using `pkg-config` or setting environment variable `PYMTL_VERILATOR_INCLUDE_DIR`. You may choose either one that you feel is the most convenient.

Using `pkg-config`

Install `pkg-config` and verify that it is setup correctly as follows:

```
$ sudo apt-get install pkg-config
$ pkg-config --variable=includedir verilator
```

If the output is a valid path to the include directory, you are all set. Otherwise you may need to refer to the next section to set up the environment variable.

Using environment variable

If `pkg-config` is not able to provide the information of Verilator, environment variable `PYMTL_VERILATOR_INCLUDE_DIR` needs to point to the include directory of the installed Verilator. If you installed Verilator to the default path, the following command will set up the variable. Replace the default path with your custom include path if necessary.

```
$ export PYMTL_VERILATOR_INCLUDE_DIR="/usr/local/share/verilator/include"
```

1.2 Quick Start Guide

This guide demonstrates how to perform arithmetics on the PyMTL3 Bits objects and how to simulate simple hardware components like a full adder and a register incrementer. All contents of this guide can be finished in a Python >= 3.6 interative REPL.

1.2.1 Bits arithmetics

Let's start with a simple addition of two 4-bit numbers. The following code snippet imports the basic PyMTL3 functionalities and creates two 4-bit objects `a` and `b`:

```
>>> from pymtl3 import *
>>> a = Bits4(4)
Bits4(0x4)
>>> b = Bits4(3)
Bits4(0x3)
```

The Bits objects support common arithmetics and comparisons:

```
>>> a + b
Bits4(0x7)
>>> a - b
Bits4(0x1)
>>> a * b
Bits4(0xC)
>>> a & b
Bits4(0x0)
>>> a | b
Bits4(0x7)
>>> a > b
Bits1(0x1)
>>> a < b
Bits1(0x0)
```

1.2.2 Full adder example

Next we will experiment with a full adder. It has already been included in the PyMTL3 code base and we can simply import it to use it in the REPL.

```
>>> from pymtl3.examples.ex00_quickstart import FullAdder
```

Thanks to the inspection feature of Python we can easily print out the source code of the full adder. You can see that the full adder logic is implemented inside an update block `upblk`.

```
>>> import inspect
>>> print(inspect.getsource(FullAdder))
class FullAdder( Component ):
  def construct( s ):
    s.a    = InPort()
    s.b    = InPort()
    s.cin  = InPort()
    s.sum  = OutPort()
    s.cout = OutPort()
```

(continues on next page)

(continued from previous page)

```
    @update
    def upblk():
      s.sum  @= s.cin ^ s.a ^ s.b
      s.cout @= ( ( s.a ^ s.b ) & s.cin ) | ( s.a & s.b )
```

To simulate the full adder, we need to apply the `DefaultPassGroup` PyMTL pass. Then we can set the value of input ports and simulate the full adder by calling `fa.sim_tick`:

```
>>> fa = FullAdder()
>>> fa.apply( DefaultPassGroup() )
>>> fa.sim_reset()
>>> fa.a @= 0
>>> fa.b @= 1
>>> fa.cin @= 0
>>> fa.sim_tick()
```

Now let's verify that the full adder produces the correct result:

```
>>> assert fa.sum == 1
>>> assert fa.cout == 0
```

1.2.3 Register incrementer example

Similar to the full adder, we can do the following to import the register incrementer component and print out its source:

```
>>> from pymtl3.examples.ex00_quickstart import RegIncr
>>> print(inspect.getsource(RegIncr))
```

And to simulate an 8-bit register incrementer:

```
>>> regincr = RegIncr( 8 )
>>> regincr.apply( DefaultPassGroup() )
>>> regincr.sim_reset()
>>> regincr.in_ @= 42
>>> regincr.sim_tick()
```

Now verify the registered output is indeed incremented:

```
>>> assert regincr.out == 43
```

CHAPTER 2

Tutorial

Tutorials for register-transfer-level and gate-level modeling.

- **Hardware modeling**: *Gate-level modeling* | *Register-transfer-level modeling*

2.1 Gate-Level Modeling

2.2 Register-Transfer-Level Modeling

CHAPTER 3

Reference

3.1 Hardware Data Types

3.1.1 Bits Type

Semantics

This section defines the expected behavior of different operations on Bits objects and Python `int` objects.

Here are the general principles of Bits/`int` operation semantics:

- Implicit truncation is not allowed.
- Implicit zero extension is allowed and will be attempted in cases of bitwidth mismatch.
- Currently all operations of bits and integers preserve the unsigned semantics.

Following the above principles, here is how to determine the bitwidth of an expression:

- A `BitsN` object has an explicit bitwidth of `N`.
- An `int` object has an inferred bitwidth of the minimal number of bits required to hold its value.
- For binary operations that are not << and >>
  - If both sides have explicit bitwidth
    - it is an error if bitwidth of both sides mismatch.
    - the result has an explicit bitwidth indicated in the operation bitwidth rule table.
  - If both sides have inferred bitwidth
    - the shorter side will be zero-extended to have the same bitwidth as the longer side.
    - the result has an inferred bitwidth indicated in the operation bitwidth rule table.
  - If one side has explicit bitwidth and the other has inferred bitwidth
    - it is an error if the inferred bitwidth is smaller than the explicit bitwidth.

    - otherwise, a zero extension on the inferred side to the explicit bitwidth is attempted.
    - the result has an explicit bitwidth indicated in the operation bitwidth rule table.

- For binary operations << and >>
  - The bitwidth of the right hand side is ignored.
  - If the left hand side has explicit bitwidth, then the result has an explicit bitwidth indicated in the operation bitwidth rule table.
  - If the left hand side has inferred bitwidth, then the result has an inferred bitwidth indicated in the operation bitwidth rule table.
- For unary operations
  - If the operand has explicit bitwidth, then the result has an explicit bitwidth indicated in the operation bitwidth rule table.
  - If the operand has inferred bitwidth, then the result has an inferred bitwidth indicated in the operation bitwidth rule table.

Opeartion bitwidth rules

Assuming the bitwidth of Bits or integer objects `i`, `j`, and `k` are `n`, `m`, and `p`, respectively. The following table defines the result of different operations. Note that the Verilog rules only apply to self-determined expressions.

Table 1: PyMTL3 and Verilog Bitwidth Rules

PyMTL Expression	Verilog Expression	PyMTL3	Verilog
i+j	i+j	max(n, m)	max(n, m)
i-j	i-j	max(n, m)	max(n, m)
i*j	i*j	max(n, m)	max(n, m)
i/j	i/j	max(n, m)	max(n, m)
i%j	i%j	max(n, m)	max(n, m)
i&j	i&j	max(n, m)	max(n, m)
i|j	i|j	max(n, m)	max(n, m)
i^j	i^j	max(n, m)	max(n, m)
~i	~i	n	n
i>>j	i>>j	n	n
i<<j	i<<j	n	n
i==j	i==j	1	1
i and j	i&&j	max(n, m)	1
i or j	i||j	max(n, m)	1
i>j	i>j	1	1
i>=j	i>=j	1	1
i<j	i<j	1	1
i<=j	i<=j	1	1
reduce_and(i)	&i	1	1
reduce_or(i)	|i	1	1
reduce_xor(i)	^i	1	1
j if i else k	i?j:k	m where m == p	max(m, p)
conat(i,..,j)	{i,. . . ,j}	n + . . . + m	n + . . . + m
i[j]	i[j]	1	1
i[j:k]	i[j:k]	k-j	k-j

Supported Bits opeartions

We recommend not using Python `and` and `or` operators on Bits/`int` objects because they carry special Python semantics which is not compliant with the PyMTL semantics. Use `&` and `|` instead.

class pymtl3.datatypes.PythonBits.**Bits**(*nbits*, *v=0*, *trunc_int=False*)

__add__(*other*)

__and__(*other*)

__bool__()

__deepcopy__(*memo*)

__eq__(*other*)
Return self==value.

__floordiv__(*other*)

__ge__(*other*)
Return self>=value.

__getitem__(*idx*)

__gt__(*other*)
Return self>value.

__hash__()
Return hash(self).

__ilshift__(*v*)

__imatmul__(*v*)

__index__()

__init__(*nbits*, *v=0*, *trunc_int=False*)
Initialize self. See help(type(self)) for accurate signature.

__int__()

__invert__()

__le__(*other*)
Return self<=value.

__lshift__(*other*)

__lt__(*other*)
Return self<value.

__mod__(*other*)

__module__ = 'pymtl3.datatypes.PythonBits'

__mul__(*other*)

__or__(*other*)

__radd__(*other*)

__rand__(*other*)

__repr__()
Return repr(self).

__rfloordiv__(*other*)

__rmod__(*other*)

__rmul__(*other*)

__ror__(*other*)

__rshift__(*other*)

__rsub__(*other*)

__rxor__(*other*)

__setitem__(*idx*, *v*)

__slots__ = ('_nbits', '_uint', '_next')

__str__()
Return str(self).

__sub__(*other*)

__xor__(*other*)

_flip()

_nbits

_next

_uint

bin()

clone()

hex()

int()

nbits

oct()

to_bits()

uint()

3.1.2 BitStruct Type

To be added...

3.2 Domain-Specific Language APIs

class pymtl3.dsl.Component.**Component**

__module__ = 'pymtl3.dsl.Component'

static __new__(*cls*, **args*, ***kwargs*)
Convention: variables local to the object is created in __new__

add_component_by_name(*name*, *obj*, *provided_connections=[]*)

add_connection(*o1*, *o2*)

add_connections(**args*)

apply(*pass_instance*)

check()

delete_component(*name*)

elaborate()

get_all_components()

get_all_explicit_constraints()

get_all_method_nets()

get_all_object_filter(*filt*)

get_all_upblk_metadata()

get_all_update_blocks()

get_all_update_ff()

get_all_update_once()

get_all_value_nets()

get_child_components(*sort_key=None*)

get_component_level()

get_connect_order()

get_input_value_ports(*sort_key=None*)

get_local_object_filter(*filt*, *sort_key=None*)

get_metadata(*key*)

Get the metadata `key` of the given component.

Can be called before, during, or after elaboration.

Parameters **key** (*MetadataKey*) – Key of the metadata.

Returns The metadata of the given `key`.

Return type object

Raises

- `TypeError` – Raised if `key` is not an instance of `MetadataKey`.
- `UnsetMetadataError` – Raised if the component does not have metadata for the given `key`.

get_output_value_ports(*sort_key=None*)

get_signal_adjacency_dict()

get_upblk_metadata()

get_update_block_host_component(*blk*)

get_update_block_info(*blk*)

get_update_block_order()

get_update_blocks()

get_update_ff()

get_wires(*sort_key=None*)

has_metadata(*key*)
Check if the component has metadata for `key`.

Can be called before, during, or after elaboration.

Parameters **key** (*MetadataKey*) – Key of the metadata.

Returns Whether or not the component has the metadata for `key`.

Return type bool

Raises `TypeError` – Raised if `key` is not an instance of `MetadataKey`.

replace_component(*foo*, *cls*, *check=True*)

replace_component_with_obj(*foo*, *new_obj*, *check=True*)

set_metadata(*key*, *value*)
Set the metadata `key` of the given component to be `value`.

Can be called before, during, or after elaboration.

Parameters

- **key** (*MetadataKey*) – Key of the metadata.
- **value** (*object*) – The metadata. Can be any object.

3.3 Hardware Modeling Primitives

3.4 Passes

PyMTL Passes are modular Python programs that can be applied on PyMTL components to analyze, instrument, or transform the given component hierarchy.

3.4.1 Communicate with passes using metadata

Metadata are per-component data that can be used to customize the behavior of various passes. PyMTL components provide two APIs to set and retrieve metadata: *Component.set_metadata(key, value)* and *Component.get_metadata(key)*, where *key* must be an instance of *MetadataKey* and *value* can be an arbitrary Python object.

PyMTL passes whose behavior can be customized are required to declare their customizable options as a *MetadataKey* class attributes of the pass. For example, you can see the list of supported options of *VerilatorImportPass* *here*. To enable the import pass on a component *m*, you can set the metadata like this

```
m.set_metadata( VerilatorImportPass.enable, True )
```

and the import pass will be able to pick up this metadata when it is applied.

If the pass you are interested in does not support customizable options or the default options can achieve what you want, you are not required to set any metadata.

3.4.2 Simulation passes

Simulation Pass API Reference

class pymtl3.passes.PassGroups.**DefaultPassGroup**(*, *vcd_file_name=None*, *textwave=False*, *print_line_trace=True*, *reset_active_high=True*)

__call__(*top*)
Call self as a function.

__init__(*, *vcd_file_name=None*, *textwave=False*, *print_line_trace=True*, *reset_active_high=True*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'pymtl3.passes.PassGroups'

class pymtl3.passes.PassGroups.**SimpleSimPass**(*debug=False*)

__call__(*top*)
Call self as a function.

__module__ = 'pymtl3.passes.PassGroups'

class pymtl3.passes.PassGroups.**AutoTickSimPass**(*print_line_trace=True*)

__call__(*top*)
Call self as a function.

__init__(*print_line_trace=True*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'pymtl3.passes.PassGroups'

API reference

3.4.3 Translation passes

PyMTL RTL Translation

For PyMTL3 RTL designs, the framework provides translation passes that translate the RTL design into various backends (e.g., Verilog). A typical PyMTL3 workflow starts from implementing and testing the DUT in the Python environment. When the designer is confident about the correctness of the DUT, he/she can simply apply the desired translation pass and use the translation result to drive an ASIC or FPGA flow.

Translatable PyMTL RTL

Translation passes only accept a translatable subset of all possible component constructs. The following constructs are translatable:

- Structural constructs
  - Data types: all Bits and BitStruct types; list of translatable data types.
  - Constants: Python integer, Bits object, and BitStruct constant instance.
  - Signals: InPort, OutPort, and Wire of Bits or BitStruct type.

  - Interfaces: all interfaces whose all child interfaces are translatable.
  - Components: all components which have translatable interfaces, signals, and update blocks.
  - List of the translatable structural constructs.
- Behavioral constructs (update blocks)
  - `@update`, `@update_ff` update blocks.
  - If-else statements
  - For-loop statements of single loop index whose range is specified through `range()`
  - Assignment statements to signals through `@=` and `<<=`
  - Assignment statements to temporary variables through `=`
  - Constants: Python integer, Bits object, and BitStruct constant instance
  - Function calls

    * `BitsN()`, `BitStruct()`, `zext()`, `sext()`, `trunc()`
  - Comparison between signals and constants
  - Arithmetic and logic operations between/on signals and constants

Some common non-translatable constructs include:

- use of common Python data structures including: set, dict, list of non-translatable constructs, etc.
- arbitrary function calls

Example: translate a PyMTL RTL component into Verilog

We will demonstrate how to translate the full adder PyMTL RTL component that is included in the PyMTL3 package. Note that if you are also interested in simulating the translated component using PyMTL3, you might find *the translate-import feature* useful because it translates the DUT and imports the translated module back for simulation.

First we need to import the full adder; the following code also dumps out how it is implemented:

```
>>> from pymtl3 import *
>>> from pymtl3.examples.ex00_quickstart import FullAdder
>>> import inspect
>>> print(inspect.getsource(FullAdder))
class FullAdder( Component ):
  def construct( s ):
    s.a    = InPort()
    s.b    = InPort()
    s.cin  = InPort()
    s.sum  = OutPort()
    s.cout = OutPort()

    @update
    def upblk():
      s.sum  @= s.cin ^ s.a ^ s.b
      s.cout @= ( ( s.a ^ s.b ) & s.cin ) | ( s.a & s.b )
```

To translate a component, it must have metadata that enables the translation pass and the translation pass needs to be applied on it. The following code shows how these can be done. The name `TranslationPass`, which refers to the Verilog translation pass, has already been included previously when we did `from pymtl3 import *`. Note

that you can set metadata of a component anytime after that object has been created, but you should only apply the translation pass after it has been elaborated.

```
>>> m = FullAdder()
>>> m.elaborate()
>>> m.set_metadata( TranslationPass.enable, True )
>>> m.apply( TranslationPass() )
```

After that you should be able to inspect the translated Verilog code in the current working directory.

```
$ less FullAdder_noparam__pickled.v
```

The string `noparam` indicates this translation result came from a PyMTL component whose `construct` method requires no arguments except for the self object `self`. If the DUT had extra arguments, those will contribute to this string. And the suffix `__pickled` indicates the translation result is a standalone Verilog source file that can be used to drive an ASIC or FPGA flow. In fact, this is true for all PyMTL translation results with the exception being your hierarchy includes a *placeholder* whose source file assumes an implicit Verilog include path.

Advanced Verilog translation

Using *the metadata mechanism*, we can customize the translation pass to have the following behaviors. See *here* for a complete list of options.

Use an explicit name for the translated top module

Assuming component *m* is the top module to be translated, the following code enforces an explicit name on the translation result of *m*

```
m.set_metadata( TranslationPass.explicit_module_name, 'FooModule' )
```

Use an explicit name for the translated .v file

Assuming component *m* is the top module to be translated, the following code enforces an explicit file name on the translated .v file

```
m.set_metadata( TranslationPass.explicit_file_name, 'FooModule.v' )
```

You can use an absolute path to dump the translation result to places other than the current working directory.

Disable components during logic synthesis in an ASIC flow

This behavior is usually desired when you want a behavioral SRAM module in the translation result but you want to disable that and swap in the real SRAM generated by a memory compiler during synthesis.

Assuming component *m* is the module to be disabled, the following code tells the translation pass to generate the appropriate Verilog code

```
m.set_metadata( TranslationPass.no_synthesis, True )
```

Also, since PyMTL3 assumes each component has implicit *clk* and *reset* pins, you can set *TranslationPass.no_synthesis_no_clk* and *TranslationPass.no_synthesis_no_reset* to *True* to remove the *clk* and *reset* pins from the module itnerfae during synthesis.

Common Verilog translation questions

Is it possbile to generate parametrized Verilog modules?

Unfortunately, the current translation mechanism relies heavily on the premise that the translated Verilog module is a design instance rather than a parametrized design. Supporting this requires non-trivial modification (even a re-design) to the translation framework and is currently not on our roadmap.

One possible workaround is to declare the desired parameters as input ports and connect these ports in the test harness or the parent module. For example, let's say we are modeling a module that takes its x-y coordinates in a mesh network as its parameters. Declaring the coordinates as parameters will lead to numerous design instances in the translation result because they are seen as different components by the translation framework. If you declare the coordinates as input ports, then there will be only one instance of the target module in the translation result because you can instantiate multiple target modules and supply different coordinates through the ports.

Translation Pass Metadata Reference

Verilog translation pass

Here are the available input and output metadata of the Verilog translation pass:

Yosys translation pass

The available input and output metadata of the Yosys translation pass are the same as those of the *Verilog translation pass*.

Introduction | *Metadata reference*

3.4.4 Import passes

External Verilog Import

PyMTL3 supports co-simulating external Verilog modules with PyMTL components with the help of Verilator. If you have *set up Verilator*, you should be able to import your Verilog designs, test their functionality with productive Python-based testing strategies, or use them as submodules of another PyMTL design.

Placeholder components

Placeholder is a property of a PyMTL *Component* which indicates that this component only declares its interface but leaves its implementation backed by external designs (i.e., modules in Verilog source files). By declaring components as having the *Placeholder* property, designers are able to concisely describe a PyMTL component hierarchy in which one or multiple components backed by external modules are interfaced to the rest of the hierarchy in a disciplined way.

Note: A *placeholder* is often used interchangably with a *placeholder component*.

Example: a top-level Verilog placeholder

Here is a full adder placeholder backed by an external Verilog module (indicated by *VerilogPlaceholder*):

```
from pymtl3 import *
class FullAdder( Component, VerilogPlaceholder ):
  def construct( s ):
    s.a = InPort()
    s.b = InPort()
    s.cin  = InPort()
    s.sum  = OutPort()
    s.cout = OutPort()
```

This placeholder only declares its interface but its implementation is supposed to be backed by external modules. You will need this kind of placeholder if you want to import an existing Verilog module for simulation.

Example: Verilog placeholder as submodule

We can further build up a new PyMTL component that has placeholder subcomponents:

```
class HalfAdder( Component ):
  def construct( s ):
    s.a = InPort()
    s.b = InPort()
    s.sum = OutPort()
    s.cout = OutPort()
    s.adder = FullAdder()
    s.adder.a //= s.a
    s.adder.b //= s.b
    s.adder.cin //= 0
    s.adder.sum //= s.sum
    s.adder.cout //= s.cout
```

The component *HalfAdder* is no longer a placeholder – it has a concrete PyMTL implementation and it's implemented using the *FullAdder* placeholder.

Import-related passes

A simulatable PyMTL component hierarchy must not have placeholders. We provide import passes to replace the placeholders in the hierarchy with simulatable components. For Verilog external modules, two passes need to be applied on the top-level component so that the hierarchy is simulatable: *pymtl3.passes.backends.verilog.VerilogPlaceholderPass* and *pymtl3.passes.backends.verilog.TranslationImportPass*. Both have been included when you do *from pymtl3 import **.

Example: import a PyMTL RTL component

We will demonstrate how to import a simple Verilog full adder. First let's make sure the Verilog file *FullAdder.v* exists under the current working directory:

```
$ echo " module FullAdder
$ (
$   input  logic clk,
$   input  logic reset,
$   input  logic a,
```

```
$   input  logic b,
$   input  logic cin,
$   output logic cout,
$   output logic sum
$ );
$   always_comb begin:
$     sum = ( cin ^ a ) ^ b;
$     cout = ( ( a ^ b ) & cin ) | ( a & b );
$   end
$ endmodule"> FullAdder.v
```

Then we will reuse the *FullAdder* placeholder in *a previous example* and apply the necessary import passes on it:

```
>>> m = FullAdder()
>>> m.elaborate()
>>> m.apply( VerilogPlaceholderPass() )
>>> m = TranslationImportPass()( m )
```

Now *m* is a simulatable component hierarchy! Let's try to feed in data through its ports...

```
>>> m.apply( DefaultPassGroup() )
>>> m.sim_reset()
>>> m.a @= 1
>>> m.b @= 1
>>> m.cin @= 1
>>> m.sim_tick()
>>> assert m.cout == 1
>>> assert m.sum == 1
```

Once we have presented data to the ports of the full adder, we invoke *sim_tick* method to evaluate the adder.

Translate-import

If you are using PyMTL3 for RTL designs, the framework also supports translating your design and importing it back for simulation (which generally happens after you have tested your design in a pure-Python environment). Since your design still exposes the same interface, you can reuse your test harness and test cases for Python simulation to test the translated Verilog design. This eliminates the need to develop Verilog test harnesses and test cases, and also enables the use of Python features for productive Verilog testing.

Example: translate-import the full adder

We will be using the full adder example from PyMTL3 in this demonstration. First let's import the design:

```
>>> from pymtl3 import *
>>> from pymtl3.examples.ex00_quickstart import FullAdder
>>> m = FullAdder()
>>> m.elaborate()
```

To translate-import this design, you will need to apply the two passes used in the previous import example. Apart from that, since we are not importing a placeholder, we also need to set up metadata to tell the translation-import pass to translate the full adder:

```
>>> m.set_metadata( TranslationImportPass.enable, True )
>>> m.apply( VerilogPlaceholderPass() )
>>> m = TranslationImportPass()( m )
```

Now we have a simulatable hierarchy backed by the translated Verilog full adder! You can find the translation result *FullAdder_no_param__pickled.v* under the current working directory. You can also simulate the adder like the following:

```
>>> m.apply( DefaultPassGroup() )
>>> m.sim_reset()
>>> m.a @= 1
>>> m.b @= 1
>>> m.cin @= 1
>>> m.sim_tick()
>>> assert m.cout == 1
>>> assert m.sum == 1
```

Advanced Verilog import

These import features make use of options offered by *VerilogPlaceholderPass* and *VerilatorImportPass*. In general, the options related to the Verilog module and source files are class attributes of *VerilogPlaceholderPass*; the options related to Verilator and C++ simulator compilation are class attributes of *VerilatorImportPass*.

While technically PyMTL is able to import any Verilatable Verilog design, code that conforms to certain rules can be imported much easier. For example, "pickled" designs are easier to import because having a standalone file saves the hassle of specifying its dependent files; we also recommend adding an *ifndef *unique_label** guard to every file to avoid potential duplicated definitions during import.

How do I specify the file name and module name of the target design?

Assuming you would like to import module *FooBar* from file *FooBarModule.v* in the same directory as the Python file that defines its wrapper *PyMTLFooBar*, you can set the *src_file* and *top_module* options on the placeholder *PyMTLFooBar*:

```
from os import path
class PyMTLFooBar( Component, Placeholder ):
  def construct( s ):
    # interface declaration here
    ...
    # Name of the top level module to be imported
    s.set_metadata( VerilogPlaceholderPass.top_module, 'FooBar' )
    # Source file path
    s.set_metadata( VerilogPlaceholderPass.src_file, path.dirname(__file__) + '/
↪FooBarModule.v' )
```

If you don't specify these two options, the default value of *top_module* will be the class name of the placeholder (e.g., *PyMTLFooBar*) and the default value of *src_file* will be *<top_module>.v* (e.g., *PyMTLFooBar.v*).

How do I specify the parameters of the target module?

Assuming you are trying to import a module with parameter *nbits = 32*. There are two ways to do that.

First you can directly add that parameter to the *construct* method of your placeholder like the following

```
class PyMTLFooBar( Component, Placeholder ):
  def construct( s, nbits ):
    # interface declaration here
    ...
```

The import pass will assume the module to be imported has a parameter named *nbits* whose value is determined during the elaboration of the PyMTL component hierarchy.

The second approach requires setting the *params* option like this:

```
class PyMTLFooBar( Component, Placeholder ):
  def construct( s, pymtl_nbits ):
    # interface declaration here
    ...
    s.set_metadata( VerilogPlaceholderPass.params, {
      'nbits' : pymtl_nbits
    } )
```

The *params* option takes a Python dictionary that maps parameter names (strings) to integers. When both *construct* arguments and the *params* option are present, the import pass prioritizes the explicit *params* option over *construct* arguments.

What if my target module does not have clk/reset pins?

PyMTL3 assumes each component has implicit *clk* and *reset* pins. By default, the import pass scans through the target Verilog file and tries to find code that defines a single-bit *clk* or *reset* pins. If you are importing a small design (maybe only a single module), this should work well and eliminate the need to manually specify whether your Verilog module has *clk* or *reset*.

If you wish to explicitly mark some placeholder as having or not having *clk*/*reset*, you can set the *has_clk* and *has_reset* options like this

```
class PyMTLFooBar( Component, Placeholder ):
  def construct( s ):
    # interface declaration here
    ...
    s.set_metadata( VerilogPlaceholderPass.has_clk, False )
    s.set_metadata( VerilogPlaceholderPass.has_reset, True )
```

The explicit *has_clk* and *has_reset* options have priority over the values inferred from the Verilog source file.

What if my target module requires a Verilog include path?

You can set the *v_include* option to a list of absolute path of include directories. Note that the current implementation only supports up to one include path.

```
from os import path
class PyMTLFooBar( Component, Placeholder ):
  def construct( s ):
    # interface declaration here
    ...
    s.set_metadata( VerilogPlaceholderPass.v_include, [path.dirname(__file__)] )
```

The above code snippet adds the directory that contains this file to the Verilog include path during import. Note that if you use placeholders with *v_include* metadata as sub-components, then during translation-import the top-level component will automatically get *v_include* metadata aggregated across all placeholders in that hierarchy.

What if my target module depends on other Verilog files?

You can set the *v_libs* option to provide a list of Verilog source files to be used together with the target source file. Suppose Verilog file *PyMTLFooBar.v* depends on *PyMTLFooBarDependency.v*, the following code snippet adds the dependency file to help Verilator resolve all definitions.

```
from os import path
class PyMTLFooBar( Component, Placeholder ):
  def construct( s ):
    # interface declaration here
    ...
    s.set_metadata( VerilogPlaceholderPass.v_libs, [path.dirname(__file__) + '/
↪PyMTLFooBarDependency.v'] )
```

Note that the files specified through *v_libs* will appear in the translation result if you translate a hierarchy that includes such placeholders.

What if the PyMTL component port names are different from Verilog port names?

You can use the *port_map* option which is a dictionary mapping ports to the name of Verilog port names (strings). The following code snippet shows how to map the PyMTL port names *in_* and *out* to Verilog port names *d* and *q*.

```
class Register( Component, Placeholder ):
  def construct( s ):
    s.in_ = InPort()
    s.out = OutPort()
    s.set_metadata( VerilogPlaceholderPass.port_map, {
      s.in_ : 'd',
      s.out : 'q',
    } )
```

How to enable Verilator coverage?

You can set option *vl_coverage*, *vl_line_coverage*, and *vl_toggle_coverage* to enable Verilator coverage (*–coverage*), line coverage (*–coverage-line*), and toggle coverage (*–coverage-toggle*).

```
class PyMTLFooBar( Component, Placeholder ):
  def construct( s ):
    # interface declaration here
    ...
    s.set_metadata( VerilatorImportPass.vl_coverage, True )
    s.set_metadata( VerilatorImportPass.vl_line_coverage, True )
    s.set_metadata( VerilatorImportPass.vl_toggle_coverage, True )
```

How to suppress certian Verilator warnings?

Here is a code snipeet that disables lint, style, and fatal warnings. It also suppresses the *MODDUP* warning. *vl_Wno_list* takes a list of warning names to be suppressed.

```
class PyMTLFooBar( Component, Placeholder ):
  def construct( s ):
    # interface declaration here
```

(continues on next page)

(continued from previous page)

```
    ...
    s.set_metadata( VerilatorImportPass.vl_W_lint, False )
    s.set_metadata( VerilatorImportPass.vl_W_style, False )
    s.set_metadata( VerilatorImportPass.vl_W_fatal, False )
    s.set_metadata( VerilatorImportPass.vl_Wno_list, [ 'MODDUP' ] )
```

How to dump VCD from the Verilator-Python co-simulation?

Here is a code snipeet that enables VCD dumping to *DUT.vcd*, sets time scale to *1ps*, sets the *clk* pin cycle to *10 * 1ps = 10ps*.

```
class PyMTLFooBar( Component, Placeholder ):
  def construct( s ):
    # interface declaration here
    ...
    s.set_metadata( VerilatorImportPass.vl_trace, True )
    s.set_metadata( VerilatorImportPass.vl_trace_filename, 'DUT.vcd' )
    s.set_metadata( VerilatorImportPass.vl_trace_timescale, '1ps' )
    s.set_metadata( VerilatorImportPass.vl_trace_cycle_time, 10 )
```

How to enable Verilog line_trace function?

If your Verilog module defines the *line_trace* function using macro *VC_TRACE_BEGIN/END*, you can enable Verilog line trace like this

```
class PyMTLFooBar( Component, Placeholder ):
  def construct( s ):
    # interface declaration here
    ...
    s.set_metadata( VerilatorImportPass.vl_line_trace, True )
```

Is it possible to add source files, include paths, or flags to the C compiler?

Note: This feature has not been thoroughly tested.

If your Verilog simulation requires external C sources, include paths, or flags, you can specify them through the following options provided by *VerilatorImportPass*: *c_flags* (string), *c_include_path* (a list of paths), *c_srcs* (a list of paths), *ld_flags* (string), *ld_libs* (string).

Common Verilog import questions

How do I import a Verilog module whose ports use a SystemVerilog bitstruct?

Your PyMTL placeholder should declare a port of the same bitwidth.

How do I import a Verilog module whose port name is *in*?

That is not supported because *in* is a Python reserved keyword. We recommend changing the port name (i.e., to *in_*).

How do I import a Verilog module with a port array?

If your module has an unpacked array port like this

```
module foo(
  input logic [31:0] foo_in [0:2][0:3]
);
  ...
endmodule
```

you will need an array of input ports like the following

```
class foo( Component, VerilogPlaceholder ):
  def construct( s ):
    # This creates a 4x3 input port array which matches the Verilog module
    s.foo_in = [ [ InPort(32) for _ in range(4) ] for _ in range(3) ]
    ...
```

If your module has a packed array port like this

```
module foo(
  input logic [2:0][3:0][31:0] foo_in
);
  ...
endmodule
```

you will need one input port of a long vector like the following

```
class foo( Component, VerilogPlaceholder ):
  def construct( s ):
    # This creates one input port whose width matches the Verilog module
    s.foo_in = InPort(3*4*32)
    ...
```

Import Pass Metadata Reference

Verilog import pass

Here are the available input and output metadata of the Verilog import pass:

Yosys import pass

The available input and output metadata of the Yosys import pass are the same as those of the *Verilog import pass*.

Introduction | *Metadata reference*

3.4.5 Translation-import passes

Translation-Import Pass Metadata Reference

Verilog translation-import pass

Here are the available input metadata of the Verilog translation-import pass:

Yosys translation-import pass

The available input metadata of the Yosys translation-import pass are the same as those of the *Verilog translation-import pass*.

Metadata reference

3.4.6 Placeholder passes

Placeholder Pass Metadata Reference

Verilog placeholder pass

class pymtl3.passes.backends.verilog.VerilogPlaceholderPass.**VerilogPlaceholderPass**(*debug=Fal*

__init__(*debug=False*)
Initialize self. See help(type(self)) for accurate signature.

__call__(*m*)
Pickle every `Placeholder` in the component hierarchy rooted at `m`.

Here are the available input metadata of the Verilog translation-import pass:

class pymtl3.passes.backends.verilog.VerilogPlaceholderPass.**VerilogPlaceholderPass**(*debug=Fal*

params = <pymtl3.dsl.MetadataKey.MetadataKey object>
A dict that maps the module parameters to their values.

Type: `{ str : int }`; input

Default value: `{}`

port_map = <pymtl3.dsl.MetadataKey.MetadataKey object>
A dict that maps the PyMTL port name to the external source port name.

Type: `{ port : str }`; input

Default value: `{}`

separator = <pymtl3.dsl.MetadataKey.MetadataKey object>
Separator string used by name-mangling of interfaces and arrays. For example, with the default value, `s.ifc.msg` will be mangled to `ifc_msg`.

Type: `str`; input

Default value: `'_'`

src_file = <pymtl3.dsl.MetadataKey.MetadataKey object>
Path to the external source file.

Type: `str`; input

Default value: <top_module>.v

top_module = <pymtl3.dsl.MetadataKey.MetadataKey object>
Top level module name in the external source file.

Type: `str`; input

Default value: PyMTL component class name

v_flist = <pymtl3.dsl.MetadataKey.MetadataKey object>
List of Verilog source file paths.

Type: `[str]`; input

Default value: []

v_include = <pymtl3.dsl.MetadataKey.MetadataKey object>
List of Verilog include directory paths.

Type: `[str]`; input

Default value: []

v_libs = <pymtl3.dsl.MetadataKey.MetadataKey object>
List of Verilog library file paths. These files will be added to the beginning of the pikcling result.

Type: `[str]`; input

Default value: []

Metadata reference

3.5 Standard Library

CHAPTER 4

Developer Manual

To be added. . .

CHAPTER 5

About PyMTL3

Visit the PyMTL website.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Index

Symbols

A

B

C

D

E

G

H

I

N

O

P

R

S

T

U

V