ECE 6745 Complex Digital ASIC Design
Tutorial 4: PyMTL3 Hardware Modeling Framework

School of Electrical and Computer Engineering
Cornell University

revision: 2025-01-27-21-27

Contents

1. Introduction

In the lab assignments for this course, we will be using Verilog for register-transfer-level modeling and the PyMTL3 hardware modeling framework for writing functional-level models, test benches, and simulator harnesses. The previous tutorial on the Verilog hardware description language already provided a good introduction to using PyMTL3 which should be suitable for the lab assignments. However, the design projects will likely require students to write their own functional-level models, test benches, and simulator harnesses from scratch. This tutorial provides much more detail about the PyMTL3 framework by essentially reproducing what you learned in the previous tutorial but using PyMTL3. As in the previous tutorial, we will be using several open-source packages and tools: the `pytest` framework for powerful test-driven Python development; Verilator (`verilator`) for converting Verilog models into C++ source code; and Surfer/GTKWave for viewing waveforms. The PyMTL3 framework is itself open source and available on GitHub here:

- `https://github.com/pymtl/pymtl3`

We are using the `pymtl4.0-dev` branch:

- `https://github.com/pymtl/pymtl3/tree/pymtl4.0-dev`

You should feel free to browse the source code for PyMTL3 on GitHub if you want to see more how various aspects of the framework are implemented. These tools are installed and available on the `ecelinux` machines. This tutorial assumes that students have completed the Linux and Git tutorials, and also that students have a basic understanding of Python.

If you need to refresh your understanding of Python, we highly recommend working through the book by Allen Downey titled "Think Python: How to Think Like a Computer Scientist" (O'Reilly, 2014). We also recommend reading a recent research paper on PyMTL3 by Shunning Jiang et al. titled "PyMTL3: A Python Framework for Open-Source Hardware Modeling, Generation, Simulation, and Verification" and published in IEEE Micro. Both of these resources are available on the course website.

Before you begin, make sure that you have **logged into the `ecelinux` servers** using VS Code as described in the remote access tutorial. To follow along with the tutorial, type the commands without the `%` character (for the `bash` prompt) or the `>>>` characters (for the `python` interpreter prompt). In addition to working through the commands in the tutorial, you should also try the more open-ended tasks marked with the ★ symbol.

Before you begin, make sure that you have **sourced the setup-ece6745.sh script**, and then clone the tutorial repository from GitHub.

```
% source setup-ece6745.sh
% mkdir -p ${HOME}/ece6745
% cd ${HOME}/ece6745
% git clone git@github.com:cornell-ece6745/ece6745-tut4-pymtl.git tut4
% cd tut4/sim
% TUTROOT=${PWD}
```

> **NOTE:** It should be possible to experiment with this tutorial even if you are not enrolled in the course and/or do not have access to the course computing resources. All of the code for the tutorial is located on GitHub. You will not use the `setup-ece6745.sh` script, and your specific environment may be different from what is assumed in this tutorial.

2. PyMTL3 for Functional-, Cycle-, and Register-Transfer-Level Modeling

Computer architects can model systems at various levels of abstraction including at the: functional-level (FL), cycle-level (CL), and register-transfer-level (RTL). In this section, we provide a brief overview of these different levels of modeling and also provide more detail on the difference between synthesizable and non-synthesizable RTL modeling.

2.1. Comparison of FL, CL, and RTL Modeling

Each level of modeling has its own unique advantages and disadvantages, so the most effective designers uses a mix of these modeling levels as appropriate. This tutorial will use various examples to illustrate how to incrementally refine a design through FL, CL, and RTL models. Although it is useful for students to understand CL modeling (and indeed most computer architects focus primarily on CL modeling), the actual lab assignments will focus on FL and RTL modeling.

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.

In this tutorial, FL, CL, and RTL models all use port-based interfaces, concurrent blocks, and structural composition. Note that PyMTL3 supports more advanced polymorphic interface connection which directly connects interfaces at different levels by automatically inserting adapters. Both the port-based approach and the polymorphic approach enable PyMTL3 to support mixed-level modeling, i.e., combining FL, CL, and RTL models of various subsystems into a single unified system model.

2.2. Synthesizable vs. Non-Synthesizable RTL Modeling

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. When using PyMTL3 for RTL modeling we need to be very careful to use only the subset of PyMTL3 which can be translated to Verilog. However, in this course students' design work will primarily be in Verilog, and we will primarily be using PyMTL3 for writing functional-level modeling, test benches, and simulator harnesses. Students can use any Python construct they like in their functional-level models, test benches, and simulator harnesses.

3. PyMTL3 Basics: Data Types and Operators

We will begin by writing some very basic code to explore PyMTL3 data types and operators. We will not be modeling actual hardware yet; we are just experimenting with the framework. Start by launching the Python interpreter and importing the PyMTL3 framework into the global namespace.

```
% mkdir -p ${TUTROOT}/build
% cd ${TUTROOT}/build
% python
>>> from pymtl3 import *
```

3.1. `Bits` Data Type

To understand any new modeling framework we usually start by exploring the primitive data types for representing values in a model. PyMTL3 uses the `Bits` class to represent fixed-bitwidth values. Note that in many hardware description languages (HDLs) each bit can take on one of four values (i.e., 0, 1, X, Z), where X is used to represent unknown values and Z is used to represent high-impedence values. In PyMTL3 each bit can only take on one of two values (i.e., 0, 1). We say that these other HDLs support *four-state values* while PyMTL3 supports *two-state values*. Both approaches have advantages and disadvantages. Two-state values produces faster simulations and avoid many of the pitfalls of using X values; but some hardware constructs are a bit more verbose to describe when only two-state values are available. Using two-state values also raises issues with properly handling reset logic, although there are well-known techniques to address these issues.

Figure 1 shows an example session in the Python interpreter that illustrates how to instantiate and manipulate `Bits` objects. Type the commands into the Python interpreter and observe the output.

PyMTL3 already offers common Bits types from `Bits1-255`. A `BitsN` constructor takes one argument specifying the initial value. Remember that in Python, a *variable* is just a *name* that refers to a *value* or *object*. So on line 2, we create a new variable with the name `a` that refers to a new `Bits16` object and an initial value of 37. Also recall that values and objects belong to different types, and that the type of a variable is the type of the value or object it refers to. As shown on line 4, the type of `a` is `Bits16`. We might also say that `a` holds an *instance* of type `Bits16`. Lines 9–11 show what happens if we assign a new integer value to the name `a`. It does not update the `Bits` object but instead simply updates the name `a` to now refer to a plain integer value 47. Lines 13–15 shows an alternative (and possibly more succinct) way of using `bN` types to create constants. Line 17–24 shows how to create wider `Bits` types by using `mk_bits(N)`. `mk_bits` can also be used to create `Bits` types that must be derived from some statically unknown bitwidth.

Lines 26–30 show how to use standard Python syntax to specify numeric literals in binary or hexadecimal form. Lines 32–38 demonstrate that negative initial values are also possible. These negative values are stored using two's complement. The `Bits` constructor includes dynamic range checking and will throw an exception if the given literal value cannot be stored using the given number of bits. Lines 40–45 illustrate one example where 300 is too large to be stored in just eight bits. Lines 47–49 illustrate the optional `Bits` constructor `trunc_int` argument that will truncate initial values which are too large to store in the given number of bits. Lines 51–58 shows how to extract number of bits and the unsigned/signed integer value from a `Bits` object.

★ *To-Do On Your Own:* Experiment with creating `Bits` objects of different bitwidths and various initial values. Experiment with the `trunc` argument to truncate large initial values.

```
 1  # BitsN takes an initial value
 2  >>> a = Bits16( 37 )
 3  >>> type(a)
 4  <class 'pymtl3.datatypes.bits_import.Bits16'>
 5  >>> a
 6  Bits16(0x0025)
 7  >>> str(a)
 8  '0025'
 9  >>> a = 47
10  >>> type(a), a
11  (<class 'int'>, 47)
12
13  # bN recommended for creating constants
14  >>> b16(37)
15  Bits16(0x0025)
16
17  # Creating wider Bits types
18  >>> Bits260
19  ...
20  NameError: name 'Bits260' is not defined
21  >>> N = 260
22  >>> BitsN = mk_bits(260)
23  >>> BitsN( 37 )
24  Bits260(0x0000000...0000025)
25
26  # Using binary and hexadecimal literals
27  >>> Bits8( 0b10101100 )
28  Bits8(0xac)
29  >>> Bits32( 0xabcd0123 )
30  Bits32(0xabcd0123)
31
32  # Negative values stored in two's complement
33  >>> Bits8( -1 )
34  Bits8(0xff)
35  >>> Bits8( -2 )
36  Bits8(0xfe)
37  >>> Bits8( -128 )
38  Bits8(0x80)
39
40  # Initial values that cannot be stored with
41  # given bitwidth throw an exception
42  >>> Bits8( 300 )
43  ...
44  ValueError: Value 0x12c is too wide for Bits8!
45  (Bits8 only accepts -0x80 <= value <= 0xff)
46
47  # Truncating initial values
48  >>> Bits8( 0xdeadbeef, trunc_int=True )
49  Bits8(0xef)
50
51  # Getting number of bits and value
52  >>> a = Bits8( 128 )
53  >>> a.nbits
54  8
55  >>> a.uint()
56  128
57  >>> a.int()
58  -128
```

Figure 1: Creating `Bits` Objects

Figure 2 shows another example session in the Python interpreter that illustrates how to slice and copy `Bits` objects. Type these commands into the Python interpreter and observe the output.

`Bits` objects are sequences of bits, so we can use standard Python syntax to specify bit slices for reading or writing fields within a `Bits` object. Note that Python slices always start with the index of the first bit in the slice and end with one past the last bit in the slice. For example, the slice `a[28:32]` on line 4 produces a new four-bit `Bits` object with the most-significant four bits from `a`.

Line 20 illustrates how to create two different names that refer to the same `Bits` object. Since there is only a single `Bits` object, if we modify that object using the name `a` (line 25), then later accesses to that object using either name will reflect this change (line 27 and 29). In other words, simply assigning `a` to `b` on line 20, *does not copy the object*. To copy the object, we must create a new `Bits` object as shown on line 33.

★ *To-Do On Your Own:* Create two new `Bits` objects: one with a bitwidth of 32 and the other with a bitwidth of eight. Assign the smaller `Bits` object to the middle of the larger `Bits` object using slices. Continue to experiment with creating `Bits` objects of different bitwidths and then using slices to read and write various fields within these `Bits` objects.

```
 1  # Python slices for reading fields
 2  >>> a = Bits32( 0xabcd0123 )
 3  >>> a[28:32]
 4  Bits4(0xa)
 5  >>> a[4:24]
 6  Bits20(0xcd012)
 7
 8  # Python slices for writing fields
 9  >>> a = Bits32( 0xabcd0123 )
10  >>> a[28:32] = 0xf
11  >>> a
12  Bits32(0xfbcd0123)
13  >>> a[4:24] = 0x210cd
14  >>> a
15  Bits32(0xfb210cd3)
16
17  # Creating two names that refer to
18  # the same Bits object
19  >>> a = Bits32( 0xabcd0123 )
20  >>> b = a
21  >>> a
22  Bits32(0xabcd0123)
23  >>> b
24  Bits32(0xabcd0123)
25  >>> a[24:32] = 0x67
26  >>> a
27  Bits32(0x67cd0123)
28  >>> b
29  Bits32(0x67cd0123)
30
31  # Copying a Bits object
32  >>> a = Bits32( 0xabcd0123 )
33  >>> b = Bits32( a )
34  >>> a
35  Bits32(0xabcd0123)
36  >>> b
37  Bits32(0xabcd0123)
38  >>> a[24:32] = 0x67
39  >>> a
40  Bits32(0x67cd0123)
41  >>> b
42  Bits32(0xabcd0123)
```

Figure 2: Slicing and Copying `Bits` Objects

3.2. `Bits` Operators

Table 1 shows the `Bits` operators that we will be primarily using in this course. Note that Python supports additional operators including `/` for division, `%` for modulus, and other generic Python object manipulation functions. These operators are not translatable, so students should avoid using these operators in their RTL models.

Logical Operators	
`&`	bitwise AND
`|`	bitwise OR
`^`	bitwise XOR
`~`	bitwise NOT

Arithmetic Operators	
`+`	addition
`-`	subtraction
`*`	multiplication

Reduction Operators	
`reduce_and`	reduce via AND
`reduce_or`	reduce via OR
`reduce_xor`	reduce via XOR

Shift Operators	
`>>`	shift right
`<<`	shift left

Relational Operators	
`==`	equal
`!=`	not equal
`>`	greater than
`>=`	greater than or equals
`<`	less than
`<=`	less than or equals

Other Functions	
`sext`	sign-extension
`zext`	zero-extension
`concat`	concatenate

Table 1: `Bits` **Operators** – Obviously there are many other operations that can be used with `Bits` objects, but these are guaranteed to be translatable.

Figure 3 shows an example session in the Python interpreter that illustrates how to use basic logical and reduction operators with `Bits` objects. Type these commands into the Python interpreter and observe the output. Note that the reduction operators produce single-bit `Bits` objects.

Lines 15–20 illustrate support for implicit operand conversion. When operators are applied to a mix of `Bits` objects and standard integer values, PyMTL3 attempts to implicitly convert the standard integer values into `Bits` objects. However, as lines 22-27 suggests, performing binary arithmetics between two `Bits` objects with different bitwidths will result in type mismatch. The solution is to use `sext`, `zext` or `trunc` to match the bitwidth of the two operands, as shown later in Figure 5

★ *To-Do On Your Own:* Write a Python function that implements a full adder. It should take three one-bit `Bits` objects as operands and return a Python tuple containing two one-bit `Bits` objects corresponding to the carry out and sum bits.

Write a Python function that returns true if two `Bits` objects are equal using just the bitwise XOR operators and the reduction operators.

```
 1  # Logical operators
 2  >>> a = Bits4( 0b1010 )
 3  >>> b = Bits4( 0b1100 )
 4  >>> a & b
 5  Bits4(0x8)   # 0b1000
 6  >>> a | b
 7  Bits4(0xe)   # 0b1110
 8  >>> a ^ b
 9  Bits4(0x6)   # 0b0110
10  >>> a ^ ~b
11  Bits4(0x9)   # 0b1001
12  >>> ~a
13  Bits4(0x5)   # 0b0101
14
15  # Implicit operand conversion
16  >>> a = Bits4( 0b1010 )
17  >>> a & 0b1100
18  Bits4(0x8)   # 0b1000
19  >>> 0b1100 & a
20  Bits4(0x8)   # 0b1000
21
22  # Type mismatch errors
23  >>> a = Bits4( 0b1010 )
24  >>> a & Bits5( 0b1100 )
25  ...
26  ValueError: Operands of '&' (and) operation must
27  have matching bitwidth, but here Bits4 != Bits5.
28
29  # Reduction operators
30  >>> a = Bits8( 0b10101100 )
31  >>> reduce_and(a)
32  Bits1(0x0)
33  >>> reduce_or(a)
34  Bits1(0x1)
35  >>> reduce_xor(a)
36  Bits1(0x0)
```

Figure 3: `Bits` **Logical and Reduction Operators**

Figure 4 shows an example session in the Python interpreter that illustrates how to use the shift, arithmetic, and relational operators with `Bits` objects. Type these commands into the Python interpreter and observe the output.

Lines 3–13 illustrate left and right shift operators that can use either a standard integer value or a `Bits` object as the shift amount. The right shift operator is a logical shift and inserts zeros in the most-significant bit positions. The bitwidth of the result from a shift is always the same as the first operand to the shift operator.

Lines 17–37 illustrate addition and subtraction operators. The bitwidth of the result is always the max of the bitwidths of the two operands. These operators perform modular arithmetic. On line 20, the result of $3 + 15$ is 18 which is represented in binary as 10010 but the result is truncated to four bits. Negative numbers are converted to two's complement before performing the addition.

Lines 41–54 illustrate relational operators for comparing two `Bits` objects. The less than and greater than operators always treat the operands as unsigned.

★ *To-Do On Your Own:* Try writing some code which does a sequence of additions resulting in overflow and then a sequence of subtractions that essentially undo the overflow. For example, use an eight-bit `Bits` object to calculate `200 + 100 + 100 - 100 - 100`. Does this expression produce the expected answer even though the intermediate values overflowed?

Write a Python function that does a signed less-than comparison between two `Bits` objects of any bitwidth. You will need to use the `nbits` attribute to determine the sign bit for each `Bits` object, and handle all four cases where either operand can be positive or negative.

```
 1  # Shift operators
 2
 3  >>> a = Bits4( 0b1011 )
 4  >>> a << 2
 5  Bits4(0xc)   # 0b1100
 6  >>> a >> 2
 7  Bits4(0x2)   # 0b0010
 8
 9  >>> b = Bits4( 2 )
10  >>> a << b
11  Bits4(0xc)   # 0b1100
12  >>> a >> b
13  Bits4(0x2)   # 0b0010
14
15  # Arithmetic operators
16
17  >>> a = Bits4( 3 )
18  >>> a + 2
19  Bits4(0x5)
20  >>> a + 15
21  Bits4(0x2)
22  >>> a - 2
23  Bits4(0x1)
24  >>> a - 15
25  Bits4(0x4)
26
27  >>> b = Bits4( 2 )
28  >>> a + b
29  Bits4(0x05)
30  >>> a - b
31  Bits4(0x01)
32
33  >>> c = Bits4( -2 )
34  >>> a + c
35  Bits4(0x01)
36  >>> a - c
37  Bits4(0x05)
38
39  # Relational operators
40
41  >>> a = Bits4(3)
42  >>> b = Bits4(2)
43  >>> a == b
44  Bits1(0x0)
45  >>> a != b
46  Bits1(0x1)
47  >>> a > b
48  Bits1(0x1)
49  >>> a >= b
50  Bits1(0x1)
51  >>> a < b
52  Bits1(0x0)
53  >>> a <= b
54  Bits1(0x0)
```

Figure 4: `Bits` **Shift, Arithmetic, and Relational Operators**

Figure 5 shows an example session in the Python interpreter that illustrates functions for concatenating, zero extending, and sign extending `Bits` objects. Type these commands into the Python interpreter and observe the output.

Lines 1–7 illustrate concatenating two `Bits` objects using the `concat` function. Lines 9–14 illustrate concatenating more than two `Bits` objects. Note that one can only concatenate actual `Bits` objects as opposed to integer literals since the exact bitwidth of a decimal or hexadecimal integer literal is ambiguous.

Lines 16–28 illustrate using the `sext` and `zext` functions to sign extend and zero extend a `Bits` object to the given larger bitwidth. Lines 30–33 illustrate using the `trunc` function to truncate a `Bits` object to the given smaller bitwidth.

★ *To-Do On Your Own:* Experiment with different variations of concatenation to create interesting bit patterns.

```
 1  # Concatenation
 2  >>> a = Bits8( 0xab )
 3  >>> b = Bits12( 0xcde )
 4  >>> concat( a, b )
 5  Bits20(0xabcde)
 6  >>> concat( b, a )
 7  Bits20(0xcdeab)
 8
 9  >>> a = Bits4( 0xd )
10  >>> b = Bits12( 0xead )
11  >>> c = Bits12( 0xbee )
12  >>> d = Bits4( 0xf )
13  >>> concat( a, b, c, d )
14  Bits32(0xdeadbeef)
15
16  # Zero extension
17  >>> a = Bits4( 0xa )
18  >>> sext( a, 8 )
19  Bits8(0xfa)
20  >>> zext( a, 8 )
21  Bits8(0x0a)
22
23  # Sign extension
24  >>> a = Bits4( 0x6 )
25  >>> sext( a, 8 )
26  Bits8(0x06)
27  >>> zext( a, 8 )
28  Bits8(0x06)
29
30  # Truncation
31  >>> a = Bits8( 0xff )
32  >>> trunc( a, 3 )
33  Bits3(0x7)
```

Figure 5: `Bits` **Other Operators**

3.3. `BitStruct` Data Type

Figure 6 shows an example session in the Python interpreter that illustrates creating and using a `BitStruct` for storing a value with predefined named bit fields. Type these commands into the Python interpreter and observe the output.

Lines 2–6 define a new `BitStruct` named `Point` that represents a two-dimensional point with two four-bit fields; one for the X coordinate and one for the Y coordinate. The bit struct class must be decorated using `@bitstruct` decorator. As a quick aside, creating a bit struct using a decorator mimics the dataclass library introduced in Python 3.7. We can instantiate new `Point` objects, turn it into a compact string, read the named fields, and write the named fields. Lines 16–17 illustrate that the `to_bits()` API can pack a bit struct instance into a `Bits` object. Lines 18–19 shows that it is also possible to unpack a `Bits` object into a bit struct instance using the class method `Point.from_nbits`. Note that the order of packing/unpacking starts from the most significant bits.

Lines 21–26 define a parameterized `BitStruct` where the bit struct class name and a dictionary that contains name/bitwidth of the two coordinate fields are given as `mk_bitstruct` call arguments. This is very useful when the bitwidths are only known at runtime.

★ *To-Do On Your Own:* Create a new `BitStruct` type for holding the an RGB color pixel. The `BitStruct` should include three fields named `red`, `green`, and `blue`. Each field should be eight bits. Experiment with reading and writing these named fields.

```
 1  # Point BitStruct
 2  >>> @bitstruct
 3  ... class Point:
 4  ...   x: Bits4
 5  ...   y: Bits4
 6  ...
 7  >>> pt1 = Point(3, 4)
 8  >>> pt1
 9  Point(Bits4(0x3),Bits4(0x4))
10  >>> str(pt1)
11  '3:4'
12  >>> pt1.x
13  Bits4(0x3)
14  >>> pt1.y
15  Bits4(0x4)
16  >>> pt1.to_bits()
17  Bits8(0x34) # notice the order!
18  >>> Point.from_bits( Bits8(0x34) )
19  Point(Bits4(0x3),Bits4(0x4))
20
21  # Parameterized Point BitStruct
22  >>> nbits = 8
23  >>> PointN = mk_bitstruct( f"Point{nbits}", {
24  ...    'x': mk_bits(nbits),
25  ...    'y': mk_bits(nbits),
26  ... })
27  ...
28  >>> pt2 = PointN( 3, 4 )
29  >>> pt2
30  Point8(Bits8(0x03),Bits8(0x04))
31  >>> pt2.to_bits()
32  Bits16(0x0304)
```

Figure 6: Creating and Using `BitStruct` Objects

4. Registered Incrementer

In this section, we will create our very first PyMTL3 hardware model and then learn how to simulate, visualize, verify, reuse, parameterize, and package this model. In order to learn about PyMTL3 more deeply, we will be using PyMTL3 for actual RTL modeling even though students will primarily be using Verilog for RTL modeling. Note that the same constructs we use for RTL modeling also be used in functional-level models.

It is good design practice to usually draw some kind of picture of the hardware we wish to model before starting to develop the corresponding PyMTL3 model. This picture might be a block-level diagram, a datapath diagram, a finite-state-machine diagram, or even a control signal table; the more we can structure our code to match this diagram the more confident we can be that our model actually models what we think it does. In this section, we wish to model the eight-bit registered incrementer shown in Figure 7. In this section, you will be gradually adding code to what we provide you in the `regincr` subdirectory.

4.1. Modeling a Registered Incrementer

Figure 8 shows one way to implement the model shown in Figure 7 using PyMTL3. Every PyMTL3 file should begin with a header comment as shown on lines 1–6. The header comment identifies the primary model in the file and includes a brief description of what the model does. Reserve discussion of the actual implementation for later in the file. In general, you should attempt to keep lines in your PyMTL3 source code to less than 74 characters. This will make your code easier to read, enable printing on standard sized paper, and facilitate viewing two source files side-by-side on a single monitor.

We begin by importing the PyMTL3 framework on line 8. A PyMTL3 model is just a Python class that inherits from the `Component` base class provided by the PyMTL3 framework. A couple of comments about the coding conventions that we will be using in this course. PyMTL3 model names should always use `CamelCaseNaming`; each word begins with a capital letter without any underscores (e.g., `RegIncr`). Port names (as well as internal signal names and model instance names) should use `underscore_naming`; all lowercase with underscores to separate words. We use `in_` to name the input port because `in` is a reserved keyword in Python. Carefully group ports to help the reader understand how these ports are related. Use port names (as well as variable and module instance names) that are descriptive; prefer longer descriptive names (e.g., `write_en`) over shorter confusing names (e.g., `wen`). We usually prefer using two spaces for each level of indentation; larger indentation can quickly result in significantly wasted horizontal space. Indentation affects a Python program's semantics; so you must be consistent in how you indent blocks. This also means you cannot mix spaces and real tab characters in your source code. Our policy is to always use spaces and never insert any real tab characters in source code. Using space can also prevent a program from looking differently across different text editor settings.

The components's construct method is used to declare the port-based interface, instantiate child components, connect ports, and define concurrent blocks. This simple model does not include any child components and does not include any internal structural connectivity. Note that we diverge from standard Python coding conventions by using `s` instead of `self` to refer to the model instance in model methods. This is to reduce the non-trivial syntactic overhead of referencing ports, signals, and child components in the constructor.

Lines 18–19 declare the port-based interface for the `RegIncr` model, which in this case includes an eight-bit input port and eight-bit output port. Ports are just class attributes that refer to instances of the `InPort` or `OutPort` classes provided by the PyMTL3 framework. The constructor for these port

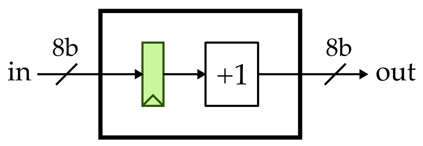


Figure 7: Block Diagram for Registered Incrementer – An eight-bit registered incrementer with an eight-bit input port, an eight-bit output port, and (implicit) clock and reset inputs.

```
 1 #=========================================================================
 2 # RegIncr
 3 #=========================================================================
 4 # This is a simple model for a registered incrementer. An eight-bit value
 5 # is read from the input port, registered, incremented by one, and
 6 # finally written to the output port.
 7
 8 from pymtl3 import *
 9
10 class RegIncr( Component ):
11
12   # Constructor
13
14   def construct( s ):
15
16     # Port-based interface
17
18     s.in_ = InPort  ( Bits8 )
19     s.out = OutPort ( Bits8 )
20
21     # update_ff block modeling register
22
23     s.reg_out = Wire( 8 ) # 8 is the same as Bits8 for Wire/InPort/OutPort
24
25     @update_ff
26     def block1():
27       if s.reset:
28         s.reg_out <<= 0
29       else:
30         s.reg_out <<= s.in_
31
32     # update block modeling incrementer
33
34     @update
35     def block2():
36       s.out @= s.reg_out + 1
```

Figure 8: Registered Incrementer – An eight-bit registered incrementer corresponding to Figure 7.

objects is parameterized by the type of values that can be sent through that port. In this example, both the input and output ports support sending eight-bit `Bits` objects. Note that we do not need to explicitly define a clock or reset input port; all PyMTL3 components have implicit `clk` and `reset` input ports. PyMTL3 components should never write the special `clk` or `reset` signal directly, and PyMTL3 components should never read the `clk` signal. PyMTL3 components can read the `reset` signal but only to reset state.

Line 23 declares an eight-bit internal wire within the model. Wires can be used to communicate values between concurrent blocks. Ports and wires are examples of PyMTL3 "signals", and for the most part we read and write all signals (i.e., both ports and wires) in the same way. Lines 25–30 define a concurrent block named `block1` to model the register in Figure 7. Concurrent blocks are just nested functions annotated with specific decorators. In this case, we use an `update_ff` decorator, which informs the framework that the corresponding nested function should be called once on every rising clock edge (i.e., the nested function should be "ticked" once per cycle). Within the nested function we refer to the implicit reset signal to determine if we should reset the `reg_out` wire to zero or copy the value on the input port to the `reg_out` wire. When writing signals from within a `update_ff` concurrent block, we always use the `<<=` operator. The `<<=` operator informs the framework that this non-blocking assignment should only be visible after all other `update_ff` concurrent blocks have executed. Using the `<<=` operator is the key to making it appear as if all `update_ff` concurrent blocks execute in parallel.

Lines 34–36 define a concurrent block named `block2` to model the combinational logic for the incrementer in Figure 7. We use the `update` decorator, which informs the framework that the corresponding nested function should be called whenever any of the signals it reads change. In this case, this means `block2` will be called whenever the value on the `reg_out` wire changes. Note that a `update` concurrent block might be called multiple times within a single clock cycle until the values read by the block reach a fixed point. If the values read by an `update` block never reach a fixed point then we say the design has a "combinational loop." When writing signals from within an `update` concurrent block, we always use the `@=` operator. Unlike using the `<<=` operator, the `@=` operator informs the framework that this blocking assignment should be visible immediately. The write to the `out` port can cause other `update` concurrent blocks in other components that read the `out` port to be called.

The two concurrent blocks work together to model the registered incrementer shown in Figure 7. On every rising clock edge, the framework will call `block1` which copies the value on the input port to the `reg_out` wire. Since `block1` is an `s.tick` concurrent block, it will appear to happen in parallel with all other `update_ff` concurrent blocks in the system. After all `update_ff` concurrent blocks have been called, the update to the `reg_out` wire will be visible. If the value on the `reg_out` wire has changed, then this will cause `block2` to be called; `block2` reads the `reg_out` wire, increments the value by one, and writes the output port. Then the whole process starts again on the next rising clock edge.

A small aside about synchronous versus asynchronous resets. Although students are allowed to read the special `reset` signal, they can only do so within a `update_ff` concurrent block (i.e., synchronous reset). Reading the reset signal in an `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.

Edit the PyMTL3 source file named `RegIncr.py` `tut4_pymtl/regincr` subdirectory using your favorite text editor. Add the combinational concurrent block shown on lines 34–36 in Figure 8 which models the incrementer logic.

4.2. Ad-Hoc Testing Using PyMTL3

Now that we have developed a new hardware model, we can test its functionality using a Python script. Figure 9 illustrates a simple Python script that elaborates the registered incrementer model, creates a simulator, writes input values to the input ports, and displays the input/output ports.

Line 12 uses a Python list comprehension to read all of the command line parameters from the `argv` variable, convert each parameter into an integer, and store these integers in a list named `input_values`. Line 16 adds three zero values to the end of the list so that our simulation will run for a few extra cycles before stopping. Lines 20–21 construct and elaborate the new `RegIncr` model. Line 25 uses the

```python
#=========================================================================
# regincr-adhoc-test <input-values>
#=========================================================================

from pymtl3  import *
from pymtl3.passes.backends.verilog import *

from sys     import argv
from RegIncr import RegIncr

# Get list of input values from command line

input_values = [ int(x,0) for x in argv[1:] ]

# Add three zero values to end of list of input values

input_values.extend( [0]*3 )

# Instantiate and elaborate the model

model = RegIncr()
model.elaborate()

# Apply the Verilog import passes and the default pass group

model.apply( VerilogPlaceholderPass() )
model = VerilogTranslationImportPass()( model )
model.apply( DefaultPassGroup() )

# Reset simulator

model.sim_reset()

# Apply input values and display output values

for input_value in input_values:

  # Write input value to input port

  model.in_ @= input_value
  model.sim_eval_combinational()

  # Print input and output ports

  print( f" cycle = {model.sim_cycle_count()}: in = {model.in_}, out = {model.out}" )

  # Tick simulator one cycle

  model.sim_tick()
```

Figure 9: Ad-Hoc Testing Using Python for Registered Incrementer – Python script to elaborate the model, apply PyMTL3 passes, write input values to the input ports, and display the input/output ports.

`DefaultPassGroup` to add simulation facilities to the top-level component. A key feature of PyMTL3 is its IMIR software architecture that separates the domain-specific language implementation (e.g., the implementation to support and collect `update_ff` and `<<=`), in-memory intermediate representation, and passes, meaning that designers create models and then apply various passes (such as the `DefaultPassGroup`,) to analyze, instrument, and transform the elaborated designs. We reset the simulator on line 29 which will raise the implicit reset signal for two cycles. Lines 33–47 define a loop that is used to iterate through the list of input values. For each input value, we write the value to the model's input port, display the values on the input/output ports, and tick the simulator. Note that we must use `@=` attribute when writing ports in the simulator script, similar to how signals are written from within `update` concurrent blocks. Otherwise, the simulator will throw an exception.

Edit the simulator script named `regincr-sim`. Add the code on lines 18–25 in Figure 9 to construct the model, elaborate the model, and build a simulator using the default pass group. Then run the simulator script as follows:

```
% cd ${TUTROOT}/build
% python ../tut4_pymtl/regincr/regincr-adhoc-test.py 0x01 0x13 0x25 0x37
```

You should see output from executing the simulator over several cycles. Note that the output starts on cycle 3; this is because calling the simulator's `reset` method raises the implicit reset signal for the first two cycles. On every cycle, we see a new input value being written into the registered incrementer, and on the *next* cycle we should see the corresponding incremented value being read from the output port.

★ *To-Do On Your Own:* Try running the simulator script with a different list of input values specified on the command line. Verify that the registered incrementer performs as expected when given the input value `0xff`.

Instead of reading the input values from the command line on line 12, experiment with generating a sequence of numbers automatically from within the script. You can use Python's `range` function to generate a sequence of numbers (potentially with a step greater than one), and you can use the `shuffle` function from the standard Python `random` module to randomly shuffle a sequence of numbers.

4.3. Visualizing a Model with Line Traces

While it is possible to visualize the execution of a model by manually inserting `print` statements both in the simulator script and in concurrent blocks, this can be quite tedious. Because this kind of visualization is so common, PyMTL3 includes built-in support for *line tracing*. A line trace consists of plain-text trace output with each line corresponding to one (and only one!) cycle. Fixed-width columns will correspond to either state at the beginning of the corresponding cycle or the output of combinational logic during that cycle. Line traces will abstract the detailed bit representations of signals in our design into useful character representations. So for example, instead of visualizing messages as raw bits, we will visualize them as text strings. Line traces can give designers a high-level view of how data is moving throughout the system.

To use line tracing, we need define a `line_trace` method in our models. Add the following method to the `RegIncr` component:

```
def line_trace( s ):
  return f"{s.in_} ({s.reg_out}) {s.out}"
```

Each component's `line_trace` method should: read the ports, wires, and other internal variables; create a fixed-width string representation of the current state and operation; and then return this string. You can use Python's extensive string manipulation capabilities to create compact and useful line traces. To display the line trace, remove the `print` statement on lines 38–39 in the `regincr-adhoc-test` script shown in Figure 9, and add `linetrace=True` as a keyword argument as follows to `DefaultPassGroup` to enable the simulator to automatically call the `line_trace` method.

```
model.apply( DefaultPassGroup(linetrace=True) )
```

Make these modifications and rerun the simulator. You can see the value at the input port, the current state of the register in the model, and the value at the output port.

★ *To-Do On Your Own:* Modify the line tracing code to show the port labels. After your modifications, the line trace might look something like this:

```
2: in:01 (00) out:01
3: in:13 (01) out:02
4: in:25 (13) out:14
```

4.4. Visualizing a Model with Text-Based Waveform

Line tracing can be useful for initially debugging the high-level behavior of your design, but it can also be useful to visualize various signals as waveforms. If you want to take a quick look at the value changes of all the signals in a small design over just a few cycles, you can display a text-based waveform *inside the terminal*.

We need to modify `regincr-adhoc-test` to enable this functionality. Pass a `textwave=True` parameter to default pass group, and add `model.print_textwave()` after the loop of `regincr-adhoc-test` script shown in Figure 9. This is because PyMTL3 doesn't want to dump the text-based waveform when your simulation is still going, so you need to call the `print_textwave` method by yourself.

```
model.apply( DefaultPassGroup(textwave=True) )
```

Make these modifications and rerun the ad-hoc test. Figure 10 shows screenshot of the terminal displaying the text-based waveform.

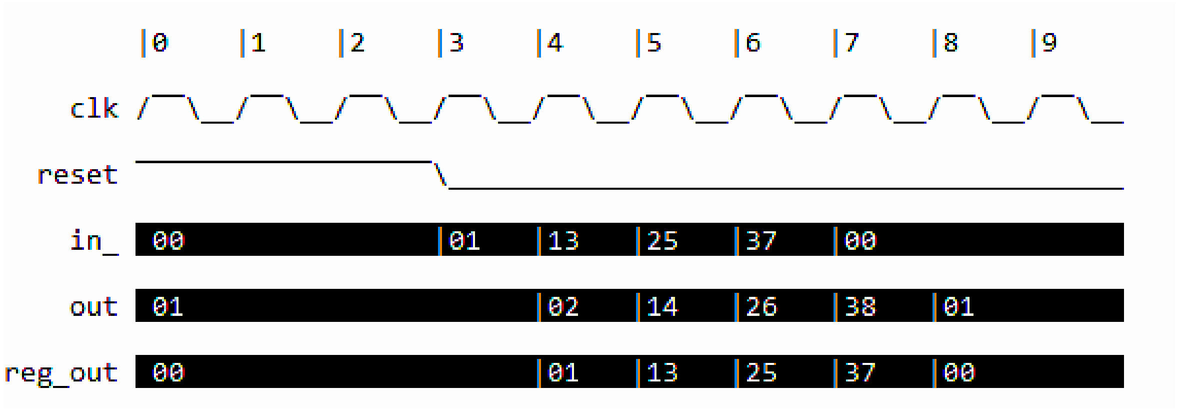
4.5. Visualizing a Model with VCD Waveforms

Line tracing can be useful for initially debugging the high-level behavior of your design, but often we need to visualize many more signals than can be easily captured in a line trace. The PyMTL3 framework can output waveforms in the Value Change Dump (VCD) format for every signal (i.e., ports and wires) in your design.

To generate VCD in `regincr-adhoc-test`, you need to pass a `vcdwave` keyword argument to the default pass group. The parameter should be a string containing the desired file name for the generated VCD (no need to add `.vcd` extension though). The default pass group will properly enable VCD dumping. Replace line 25 in the `regincr-adhoc-test` script shown in Figure 9 with the following line of code.

```
model.apply( DefaultPassGroup(vcdwave='regincr-adhoc-test) )
```

Make these modifications and rerun the ad-hoc test.



```
        |0    |1    |2    |3    |4    |5    |6    |7    |8    |9
    clk /‾‾\__/‾‾\__/‾‾\__/‾‾\__/‾‾\__/‾‾\__/‾‾\__/‾‾\__/‾‾\__/‾‾\__
  reset ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾\_________________________________________
    in_ 00                |01   |13   |25   |37   |00
    out 01                      |02   |14   |26   |38   |01
reg_out 00                      |01   |13   |25   |37   |00
```

Figure 10: PyMTL3 Text-Based Waveforms – Text-based waveforms are being used to display signals directly in the terminal associated with the registered incrementer shown in Figure 8.

```
% cd ${TUTROOT}/build
% python ../tut4_pymtl/regincr/regincr-adhoc-test.py 0x01 0x13 0x25 0x37
```

There are two options for viewing VCD files: (1) the Surfer extension for VS Code; or (2) the stand-alone GTKWave program.

Surfer only works if you are using VS Code. You just need to use the standard `code` command to open the VCD file and VS Code will automatically take care of launching Surfer.

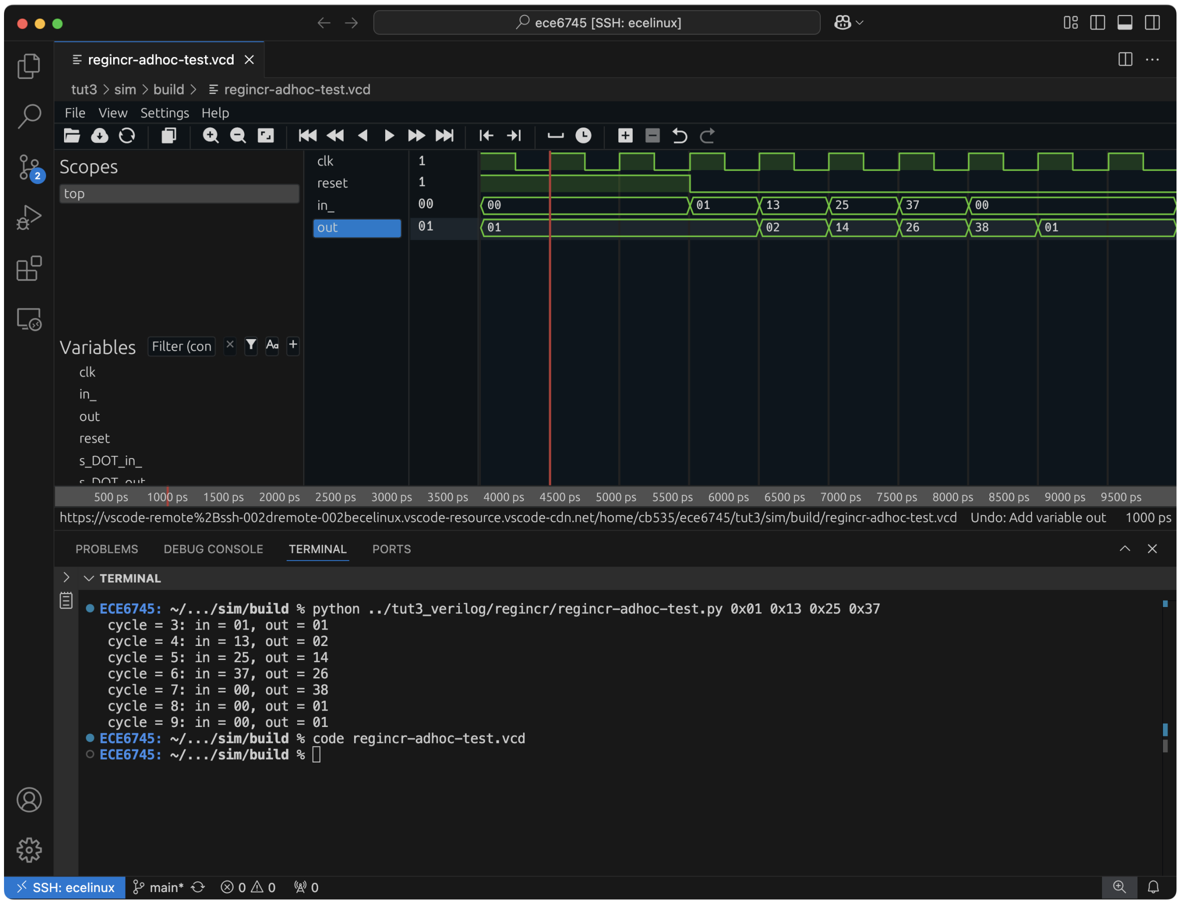
```
% cd ${TUTROOT}/build
% code regincr-adhoc-test.vcd &
```

You can browse the module hierarchy of your design in the upper-left "Scopes" panel, with the signals in any given module being displayed in the lower-left "Variables" panel. Clicking on a signal will cause it to be added to the waveform panel on the right. You can drag-and-drop signals to arrange them as desired. You can use the scrollbar at the bottom to scroll to the right through the waveform, and you can use the *View > Zoom In* and *View > Zoom Out* menu options or the corresponding magnifying glass icons in the toolbar to zoom in or out. To see the full hierarchical names of each signal choose *Settings > Variable names > Global*. Choose *File > Reload* (or click the circular arrows icon in the toolbar) to update Surfer after you have rerun a simulation. Organizing signals can sometimes be quite time consuming, so you can save and load the current configuration using *File > Save state as* and *File > Load state*. Figure 11 illustrates using Surfer to view the waveforms from our ad-hoc test. Surfer has many useful options which can make debugging your design more productive, so feel free to explore the associated documentation.

The stand-alone GTKWave program is a Linux GUI application which means you will need to be logged into `ecelinux` using Microsoft Remote Desktop. You can then use GTKWave to browse the generated waveforms by entering the following commands in the terminal within Microsoft Remote Desktop:

```
% cd ${TUTROOT}/build
% gtkwave regincr-adhoc-test.vcd &
```

You can browse the module hierarchy of your design in the upper-left panel, with the signals in any given module being displayed in the lower-left panel. Select signals and use the *Append* or *Insert*


ece6745 [SSH: ecelinux]
regincr-adhoc-test.vcd
tut3 > sim > build > regincr-adhoc-test.vcd
File View Settings Help
Scopes
top
Variables
clk
in_
out
reset
s_DOT_in_
clk 1
reset 1
in_ 00
out 01
00 01 13 25 37 00
01 02 14 26 38 01
500 ps 1000 ps 1500 ps 2000 ps 2500 ps 3000 ps 3500 ps 4000 ps 4500 ps 5000 ps 5500 ps 6000 ps 6500 ps 7000 ps 7500 ps 8000 ps 8500 ps 9000 ps 9500 ps
https://vscode-remote%2Bssh-002dremote-002becelinux.vscode-resource.vscode-cdn.net/home/cb535/ece6745/tut3/sim/build/regincr-adhoc-test.vcd Undo: Add variable out 1000 ps
PROBLEMS DEBUG CONSOLE TERMINAL PORTS
TERMINAL
SSH: ecelinux main*

```
ECE6745: ~/.../sim/build % python ../tut3_verilog/regincr/regincr-adhoc-test.py 0x01 0x13 0x25 0x37
  cycle = 3: in = 01, out = 01
  cycle = 4: in = 13, out = 02
  cycle = 5: in = 25, out = 14
  cycle = 6: in = 37, out = 26
  cycle = 7: in = 00, out = 38
  cycle = 8: in = 00, out = 01
  cycle = 9: in = 00, out = 01
ECE6745: ~/.../sim/build % code regincr-adhoc-test.vcd
ECE6745: ~/.../sim/build %
```

Figure 11: Surfer Waveform Viewer – Surfer extension for VS Code is being used to browse the signals associated with the registered incrementer shown in Figure 8 and the ad-hoc test shown in Figure 13.

button to add them to the waveform panel on the right. You can drag-and-drop signals to arrange them as desired. You can use the scrollbar at the bottom to scroll to the right through the waveform, and you can use the *Time > Zoom* menu or the corresponding magnifying glass icons in the toolbar to zoom in or out. To see the full hierarchical names of each signal choose *Edit > Toggle Trace Hierarchy* or simply press the H key. Choose *File > Reload Waveform* (or click the blue circular arrow icon in the toolbar) to update GTKWave after you have rerun a simulation. Organizing signals can sometimes be quite time consuming, so you can save and load the current configuration using *File > Write Save File* and *File > Read Save File*. Figure 12 illustrates using GTKWave to view the waveforms from our ad-hoc test. GTKWave has many useful options which can make debugging your design more productive, so feel free to explore the associated documentation.

★ *To-Do On Your Own:* Edit the register incrementer so that it now increments by +2 instead of +1. Rerun the simulator script and take another look the waveforms to see how they have changed. When you are finished, edit the registered incrementer so that it again increments by +1.

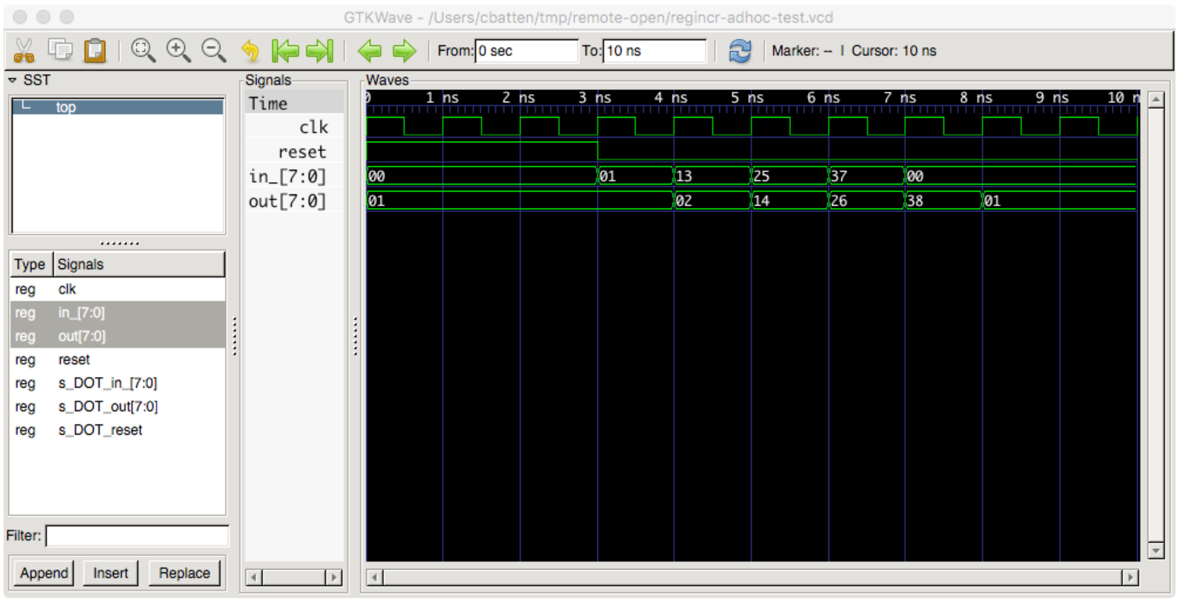


Figure 12: GTKWave Waveform Viewer – GTKWave is being used to browse the signals associated with the registered incrementer shown in Figure 8 and the ad-hoc test shown in Figure 13.

4.6. Verifying a Model with Unit Testing

Now that we have developed a new hardware model, our first thought should always turn to testing that model. Students might be tempted to simply look at line traces and/or waveforms from a simulator script to determine if their design is working, but this kind of "verification by inspection" is error prone and not reproducible. If you later make a change to your design, you would have to take another look at the line traces and/or waveforms to ensure that your design still works. If another member of your group wants to understand your design and verify that it is working, he or she would also need to take a look at the line traces and/or waveforms. While this might be feasible for very simple designs, it is obviously not a scalable approach when building the more complicated designs we will tackle in this course. Automated testing through unit testing is the best way to rigorously verify your designs.

We could simply write ad-hoc Python scripts to unit test our designs. These scripts would instantiate our design, write values to the input ports, and then verify the outputs. Unfortunately, there are many issues with using ad-hoc unit testing. Ad-hoc unit testing is usually verbose, which makes it error prone and more cumbersome to write tests. Ad-hoc unit testing is difficult for others to read and understand since by definition it is ad-hoc. Ad-hoc unit testing does not use any kind of standard test output, and does not provide support for controlling the amount of test output. In this course, we will be using the powerful `pytest` unit testing framework. The `pytest` framework is popular in the Python programming community with many features that make it well-suited for test-driven hardware development including: no-boilerplate testing with the standard `assert` statement; automatic test discovery; helpful traceback and failing assertion reporting; standard output capture; sophisticated parameterized testing; test marking for skipping certain tests; distributed testing; and many third-party plugins. More information is available at `http://www.pytest.org`.

Figure 13 illustrates a simple unit testing script for our registered incrementer. Notice at a high-level the test code is very straight-forward; the `pytest` framework enables unit testing to be as simple or as complex as necessary. The `pytest` framework includes automatic test discovery, which means that

```python
#=========================================================================
# RegIncr_simple_test
#=========================================================================

from pymtl3 import *

from pymtl3.stdlib.test_utils import config_model_with_cmdline_opts

from ..RegIncr import RegIncr

# In pytest, unit tests are simply functions that begin with a "test_"
# prefix. PyMTL3 is setup to collect command line options. Simply specify
# "cmdline_opts" as an argument to your unit test source code,
# and then you can dump VCD by adding --dump-vcd option to pytest
# invocation from the command line.

def test_basic( cmdline_opts ):

  # Create the model

  model = RegIncr()

  # Configure the model

  model = config_model_with_cmdline_opts( model, cmdline_opts, duts=[] )

  # Create and reset simulator

  model.apply( DefaultPassGroup(linetrace=True) )
  model.sim_reset()

  # Helper function

  def t( in_, out ):

    # Write input value to input port

    model.in_ @= in_

    # Ensure that all combinational concurrent blocks are called

    sim.sim_eval_combinational()

    # If reference output is not '?', verify value read from output port

    if out != '?':
      assert model.out == out

    # Tick simulator one cycle

    sim.sim_cycle()

  # Cycle-by-cycle tests

  t( 0x00, '?'  )
  t( 0x13, 0x01 )
  t( 0x27, 0x14 )
  t( 0x00, 0x28 )
  t( 0x00, 0x01 )
  t( 0x00, 0x01 )
```

Figure 13: Unit Test Script for Registered Incrementer – A unit test for the eight-bit registered incrementer in Figure 8, which uses the `pytest` unit testing framework.

it will look through the unit test script and assume that any function that begins with `test_` is a test case. In this example, `pytest` will discover a single test case named `test_basic` corresponding to the function declared on lines 16–59. To test our registered incrementer, we need to instantiate and elaborate the model, use the default pass group to add simulation facilities, write values to the input ports of the model, and finally verify that the values read from the output ports of the model are correct.

Lines 20–24 instantiate and configures the model using the command line options. Note that the `dump_vcd` flag passed from the command lines is collected in the `cmdline_opts` dictionary to the unit test. If a user includes `--dump-vcd` on the command-line when running `pytest`, then the framework will generate a VCD file for every unit test. The name of the VCD file is derived from the name of the unit test. If a user does not include `--dump-vcd` on the command-line when running `pytest`, then `dump_vcd` will be `None` and no VCD file will be generated. Lines 28–29 use the `SimulationTool` to create and reset a simulator.

Lines 33–50 define a simple helper function that is responsible for verifying one cycle of execution. The helper function takes the desired test input and the reference test output as arguments. Line 37 writes the test input to the `in_` port of the registered incrementer. Note that it is important to use `@=` operator when writing ports in the test harness, similar to how signals are written from within `update` concurrent blocks. Line 41 tells the simulator to call any `update` concurrent blocks whose input values have changed. Lines 45–46 read the `out` port and compare it to the reference output to ensure that the registered incrementer is functioning correctly. Notice that we check to make sure the reference output is not set to a question mark character. This gives us a simple way to indicate that we do not care what the output value is on that cycle. Also notice that the `pytest` framework does not need special assertion checking functions, and instead hooks into the standard `assert` statement provided in Python. This means the `pytest` framework can carefully track the `assert` statement on line 46, and on an assertion error will display the context of the `assert` statement including the sequence of function calls that lead to the assertion and the values of the variables used in the `assert` statement.

Lines 54–59 use our helper function to test the registered incrementer over six cycles. These test cases are an example of *directed cycle-by-cycle gray-box testing*. It is directed since we are explicitly creating directed tests as opposed to using some kind of random testing. It is cycle-by-cycle since we are explicitly setting the inputs and verifying the outputs every cycle. *Black-box testing* describes a testing strategy where the test cases depend only on the interface and not the specific implementation of the DUT (i.e., they should be valid for any correct implementation). *White-box testing* describes a testing strategy where the test cases depend on the specific implementation of the DUT (i.e., they may not be valid for every correct implementation). The test cases in Figure 13 are *black-box* with respect to the functional behavior of the DUT, but they are *white-box* with respect to the timing behavior of the device. The test cases rely on the fact that the registered incrementer includes exactly one edge and they would fail if we pipelined the incrementer such that each transaction took two edges. In Section 6, we will see how we can use latency-insensitive interfaces to create true black-box unit tests.

Edit the test script named `RegIncr_test.py`. Note that it is important that all test script file names end in `_test.py`, since this suffix is used by the `pytest` framework for automatic test discovery. Add the tests cases shown on lines 54–59 in Figure 8. We can run the test script using `pytest` as follows:

```
% cd ${TUTROOT}/build
% pytest ../tut4_pymtl/regincr/test/RegIncr_test.py
```

Note that we run our unit test scripts from within a separate build directory. The PyMTL3 framework often creates extra temporary and/or output files, so keeping these generated files in a separate build

```
========================= test session starts ==========================
...
collected 1 items

../tut4_pymtl/regincr/test/RegIncr_test.py .

======================= 1 passed in 0.04 seconds =======================
```

Figure 14: `pytest` **Output –** Each line corresponds to one test script, and each dot corresponds to one passing test case. Failing test cases are shown with an `F` character.

directory helps avoid creating generated files in the source tree and facilitates performing a clean build. The `pytest` framework automatically discovers the `test_basic` test case. The output from running `pytest` should look similar to what is shown in Figure 14; `pytest` will display the name of the test script and a single dot indicating that the corresponding test case has passed. If we ran multiple test scripts, then each test script would have a separate line in the output. If we had multiple `test_` functions in `RegIncr_test.py`, then each test case would have its own dot. Failing test cases are shown with an `F` character.

Note that our test script prints the line trace, yet the line trace is not included in the output shown in Figure 14. This is because by default, the `pytest` framework "captures" the standard output from a test script instead of displaying this output. The output is only displayed when a test case fails, or if the users explicitly disables capturing the standard output. So to generate a line trace for this test, we simply use the `--capture=no` (or `-s`) command line option as follows:

```
% cd ${TUTROOT}/build
% pytest ../tut4_pymtl/regincr/test/RegIncr_test.py -s
```

Note that by default, `pytest` will not show much detail on an error. This enables a designer to quickly get an overview of which tests are passing and which tests are failing. If some of your tests are failing, then you will want to produce more detailed error output using the `--tb` command line options.

```
% cd ${TUTROOT}/build
% pytest ../tut4_pymtl/regincr/test/RegIncr_test.py --tb=short
% pytest ../tut4_pymtl/regincr/test/RegIncr_test.py --tb=long
```

The `--tb` command line option specifies the level of "trace-back" output, and there are a couple of different options you might want to use including: `long`, `short`, and `line`. To generate waveforms for this test, we simply use the `--dump-vcd` command line option as follows:

```
% cd ${TUTROOT}/build
% pytest ../tut4_pymtl/regincr/test/RegIncr_test.py --dump-vcd
% gtkwave tut4_pymtl.regincr.test.RegIncr_test__test_basic.vcd &
```

★ *To-Do On Your Own:* Edit the register incrementer so that it now increments by +2 instead of +1. Rerun the unit test and verify that the tests no longer pass. Use the `--tb=long` command line option to display more detailed error output. Study the output carefully to understand the corresponding error messages. You should see: (1) a sequence of two function calls that lead to the assertion failure; (2) the exact assertion that is failing; (3) the value of the output port and the reference output in the failing assertion; and (4) the captured standard output which usually a line trace. Modify the unit test so that it includes the correct reference outputs for a +2 incrementer, rerun the unit test, and verify that the test now passes. When you are finished, edit the registered incrementer so that it again increments by +1.

4.7. Verifying a Model with Test Vectors

The unit test shown in Figure 13 requires quite a bit of setup code. Usually we want to include many directed test cases in a test script; each test case focuses on testing a different specific aspect of our design. If we simply extend the approach shown in Figure 13, then each test case would need to duplicate lines 16–50. We could refactor this code into a separate helper function that can be reused across all test cases in a given test script. However, since this kind of testing is so common, PyMTL3 includes a flexible helper function for unit testing any model using test vectors. This function is named `run_test_vector_sim` and it is part of `pymtl3.stdlib` (PyMTL3 Standard Library), which has a variety of RTL and testing functions, classes, and models that we will be using in this class. To find out more about `stdlib`, you can browse the source code on the public PyMTL3 GitHub repository:

- `https://github.com/pymtl/pymtl3/tree/pymtl4.0-dev/pymtl3/stdlib/primitive`
- `https://github.com/pymtl/pymtl3/tree/pymtl4.0-dev/pymtl3/stdlib/test_utils`

For example, here is the definition of the `run_test_vector_sim` helper function:

- `https://github.com/pymtl/pymtl3/blob/pymtl4.0-dev/pymtl3/stdlib/test_utils/test_helpers.py#L275`

Test vectors are essentially a table of test inputs and reference outputs. Figure 15 shows an extra test script that uses the `run_test_vector_sim` helper function provided by the PyMTL3 framework. There are three test cases for testing small input values, large input values, and the registered incrementer's overflow condition. The `run_test_vector_sim` helper function takes two arguments: an instantiated model and a test vector table. The function elaborates a model, uses the simulation tool to create a simulator, resets the simulator, writes the input values provided in the test vector table to the model's input ports, reads the values from the model's output ports, and compares the values to the reference values provided by the test vector table. The test vector table is a list of lists and is written so as to look like a table. Each column corresponds to either an input value or a reference output value, and each row corresponds to one cycle of the simulation. Question marks are allowed for reference output values when we don't care what the output is on that cycle. The first row of the test vector table is always a special "header string" that specifies the name of the model's input/output port for that column. Output ports are denoted with an asterisk suffix. Note how compact this test script is compared to the test script in Figure 13. This sophisticated helper function demonstrates the power of using a general-purpose dynamic language such as Python to write test harnesses.

Edit the new test script named `RegIncr_extra_test.py`. Add the code on lines 35–46 in Figure 15 which tests for overflow. Run this extra test script using `pytest` as follows:

```
% cd ${TUTROOT}/build
```

```
#=========================================================================
# RegIncr_test
#=========================================================================

from pymtl3 import *
from pymtl3.stdlib.test_utils import run_test_vector_sim
from ..RegIncr import RegIncr

#-------------------------------------------------------------------------
# test_small
#-------------------------------------------------------------------------

def test_small( cmdline_opts ):
  run_test_vector_sim( RegIncr(), [
    ('in_   out*'),
    [ 0x00, '?'  ],
    [ 0x03, 0x01 ],
    [ 0x06, 0x04 ],
    [ 0x00, 0x07 ],
  ], cmdline_opts )

#-------------------------------------------------------------------------
# test_large
#-------------------------------------------------------------------------

def test_large( cmdline_opts ):
  run_test_vector_sim( RegIncr(), [
    ('in_   out*'),
    [ 0xa0, '?'  ],
    [ 0xb3, 0xa1 ],
    [ 0xc6, 0xb4 ],
    [ 0x00, 0xc7 ],
  ], cmdline_opts )

#-------------------------------------------------------------------------
# test_overflow
#-------------------------------------------------------------------------

def test_overflow( cmdline_opts ):
  run_test_vector_sim( RegIncr(), [
    ('in_   out*'),
    [ 0x00, '?'  ],
    [ 0xfe, 0x01 ],
    [ 0xff, 0xff ],
    [ 0x00, 0x00 ],
  ], cmdline_opts )
```

Figure 15: Unit Test Script using Test Vectors for Registered Incrementer – A unit test for the eight-bit registered incrementer in Figure 8, which uses test vectors and the `pytest` unit testing framework.

```
 % pytest ../tut4_pymtl/regincr/test/RegIncr_extra_test.py
```

The output should show the name of the test script and three dots corresponding to the three test cases in Figure 15. The `pytest` framework can automatically discover test scripts in addition to automatically discovering the test cases within a test script. If the argument to `pytest` is a directory, then `pytest` will search that directory for any files ending in `_test.py` and assume that these files are

```
========================= test session starts ==========================
...
collected 21 items

../tut4_pymtl/regincr/test/RegIncr2stage_test.py::test_small FAILED
../tut4_pymtl/regincr/test/RegIncr2stage_test.py::test_large FAILED
../tut4_pymtl/regincr/test/RegIncr2stage_test.py::test_overflow FAILED
../tut4_pymtl/regincr/test/RegIncr2stage_test.py::test_random FAILED
../tut4_pymtl/regincr/test/RegIncrNstage_test.py::test[2stage_small] FAILED
../tut4_pymtl/regincr/test/RegIncrNstage_test.py::test[2stage_large] FAILED
../tut4_pymtl/regincr/test/RegIncrNstage_test.py::test[2stage_overflow] FAILED
../tut4_pymtl/regincr/test/RegIncrNstage_test.py::test[2stage_random] FAILED
../tut4_pymtl/regincr/test/RegIncrNstage_test.py::test[3stage_small] FAILED
../tut4_pymtl/regincr/test/RegIncrNstage_test.py::test[3stage_large] FAILED
../tut4_pymtl/regincr/test/RegIncrNstage_test.py::test[3stage_overflow] FAILED
../tut4_pymtl/regincr/test/RegIncrNstage_test.py::test[3stage_random] FAILED
../tut4_pymtl/regincr/test/RegIncrNstage_test.py::test_random[1] PASSED
../tut4_pymtl/regincr/test/RegIncrNstage_test.py::test_random[2] FAILED
../tut4_pymtl/regincr/test/RegIncrNstage_test.py::test_random[3] FAILED
../tut4_pymtl/regincr/test/RegIncrNstage_test.py::test_random[4] FAILED
../tut4_pymtl/regincr/test/RegIncrNstage_test.py::test_random[5] FAILED
../tut4_pymtl/regincr/test/RegIncrNstage_test.py::test_random[6] FAILED
../tut4_pymtl/regincr/test/RegIncr_simple_test.py::test_basic PASSED
../tut4_pymtl/regincr/test/RegIncr_test.py::test_small PASSED
../tut4_pymtl/regincr/test/RegIncr_test.py::test_large PASSED

================== 17 failed, 4 passed in 0.36 seconds ==================
```

Figure 16: `pytest` **Verbose Output –** Each line corresponds to one test case. Passing test cases are marked with `PASSED` and failing test cases are marked with `FAILED`.

test scripts. The `pytest` framework also provides a more verbose output where each test case is listed on a separate line; passing test cases are marked with `PASSED` and failing test cases are marked with `FAILED`. Run both of the test scripts using the `--verbose` (or `-v`) command line option as follows:

```
% cd ${TUTROOT}/build
% pytest ../tut4_pymtl/regincr/test -v
```

The verbose output should look similar to what is shown in Figure 16. Some test cases are passing for those models which we have completed, while other test cases are failing because we will work on them later in the tutorial. We can use the `-k` command line option to select just a few test cases to run and debug in more detail. For example to run just the test case for testing small input values, we can use the following:

```
% cd ${TUTROOT}/build
% pytest ../tut4_pymtl/regincr/test -k small
```

We can use the `-x` command line option to have `pytest` stop after the very first failing test case:

```
% cd ${TUTROOT}/build
% pytest ../tut4_pymtl/regincr -x
```

When testing an entire directory, we often use an iterative process to “zoom” in on a failing test case. We start by running all tests in the directory to see an overview of which tests are passing and which tests are failing. We then explicitly run a single test script with the `-v` command line option to see which specific test cases are failing. Finally, we use the `-k` or `-x` command line options with `--tb`, `-s`, and/or `--dump-vcd` command line option to generate error output, line traces, and/or waveforms

```
#-------------------------------------------------------------------------
# test_random
#-------------------------------------------------------------------------

import random

def test_random( cmdline_opts ):

  test_vector_table = [( 'in_', 'out*' )]
  last_result = '?'
  for i in range(20):
    rand_value = Bits8( random.randint(0,0xff) )
    test_vector_table.append( [ rand_value, last_result ] )
    last_result = Bits8( rand_value + 1, trunc_int=True )

  run_test_vector_sim( RegIncr(), test_vector_table, cmdline_opts )
```

Figure 17: Random Test Case for Registered Incrementer – Random input values and the corresponding incremented output value are added to a test vector table for random testing.

for the failing test case. Here is an example of this three-step process to "zoom" in on a failing test case:

```
% cd ${TUTROOT}/build
% pytest ../tut4_pymtl/regincr/test/
% pytest ../tut4_pymtl/regincr/test/RegIncr2stage_test.py -v
% pytest ../tut4_pymtl/regincr/test/RegIncr2stage_test.py -v -x --tb=short
% pytest ../tut4_pymtl/regincr/test/RegIncr2stage_test.py -v -x --tb=long
```

★ *To-Do On Your Own:* Add another directed test case for the registered incrementer which tests another arbitrary set of input values. Rerun the test script, and verify that the output matches your expectations.

4.8. Verifying a Model with Random Testing

So far we used a directed cycle-by-cycle gray-box testing strategy. Once we have finished writing hand-crafted directed tests, we almost always want to leverage randomized testing to further improve our confidence in the correct functionality of the design. Generating random test vectors in Python is relatively straight forward, especially if we make use of the standard Python `random` module. Figure 17 illustrates a random test case for the registered incrementer. Note that the random test vector generation must carefully take into account the latency of the registered incrementer in order to ensure that each reference output is placed in the correct row of the test vector table. Add this test case to the `RegIncr_extra_test.py` test script, and run the new test case with line tracing enabled as follows:

```
% cd ${TUTROOT}/build
% pytest ../tut4_pymtl/regincr/test/RegIncr_extra_test.py -k random -s
```

★ *To-Do On Your Own:* Add another random test case for the registered incrementer where the input values are always less than 16 (i.e., small numbers). Rerun the test script, and verify that the output matches your expectations.

4.9. Reusing a Model with Structural Composition

We will use modularity and hierarchy to structurally compose small, simple models into large, complex models. This incremental approach allows us to first design and test the small models, and thus ensure they are working, before integrating them and testing the larger models. Figure 18 shows a two-stage registered incrementer that uses structural composition to instantiate and connect two instances of a single-stage registered incrementer. Figure 19 shows the corresponding PyMTL3 model. Line 9 imports the child model that we will be reusing.

Lines 19–20 illustrate a simplified PyMTL3 syntax for specifying the type of the values that can be passed through the `in_` and `out` ports. If we use an integer `N`, then this is syntactic sugar for specifying that objects of type `BitsN` can be passed through the port.

Lines 24–33 actually perform the structural composition of the two instances of the child model. Line 24 instantiates the first `RegIncr` model with the instance name `reg_incr_0`. Line 26 uses the `connect` function to connect two ports together: the `in_` port, which is part of the parent interface, and the `in_` port for the first `RegIncr`. The arguments to the `connect` method can be ports or wires and can be in either order (i.e., the input signal is not required to be the first argument). Line 30 instantiates the second `RegIncr` model with the instance name `reg_incr_1`. Line 32 connects the output of the first `RegIncr` to the input of the second `RegIncr`. Line 33 connects the output of the second `RegIncr` to the `out` port in the parent interface using //= operator which is a syntactic sugar for `connect`. Ever since //= operator was introduced, almost all PyMTL3 design code has been using //= for connections.

Lines 37–43 show the `line_trace` method for the two-stage registered incrementer. A key feature of line tracing is the ability to construct line trace strings hierarchically. On lines 40–41, we call the `line_trace` methods for the two child `RegIncr` models.

As always, once we create a new hardware model, we should immediately write a unit test to verify its functionality. Figure 20 shows a test script using test vectors to verify our two-stage registered incrementer. Notice how we must carefully take into account the two-cycle latency of the registered incrementer in order to ensure that each reference output is placed in the correct row of the test vector table. This is because we are using a cycle-by-cycle gray-box testing strategy.

Edit the PyMTL3 source file named `RegIncr2stage.py`. Add lines 28-33 from Figure 19 to connect the second stage of the two-stage registered incrementer. Then run all of the test scripts as well as a subset of the test cases as follows:

```
% cd ${TUTROOT}/build
% pytest ../tut4_pymtl/regincr/test/RegIncr2stage_test.py -v
```

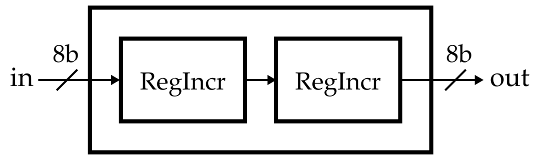


Figure 18: Block Diagram for Two-Stage Registered Incrementer – An eight-bit two-stage registered incrementer that reuses the registered incrementer in Figure 7 through structural composition.

```
#=========================================================================
# RegIncr2stage
#=========================================================================
# Two-stage registered incrementer that uses structural composition to
# instantiate and connect two instances of the single-stage registered
# incrementer.

from pymtl3 import *

from tut4_pymtl.regincr.RegIncr import RegIncr

class RegIncr2stage( Component ):

  # Constructor

  def construct( s ):

    # Port-based interface

    s.in_ = InPort (8)
    s.out = OutPort(8)

    # First stage

    s.reg_incr_0 = RegIncr()

    connect( s.in_, s.reg_incr_0.in_ )

    # Second stage

    s.reg_incr_1 = RegIncr()

    s.reg_incr_0.out //= s.reg_incr_1.in_
    s.reg_incr_1.out //= s.out

  # Line Tracing

  def line_trace( s ):
    return "{} ({}|{}) {}".format(
      s.in_,
      s.reg_incr_0.line_trace(),
      s.reg_incr_1.line_trace(),
      s.out
    )
```

Figure 19: Two-Stage Registered Incrementer – An eight-bit two-stage registered incrementer corresponding to Figure 18. This model is implemented using structural composition to instantiate and connect two instances of the single-stage register incrementer.

```
% pytest ../tut4_pymtl/regincr/test/RegIncr2stage_test.py -k test_small
```

You can generate the line trace for just the first test case for our two-stage registered incrementer as follows:

```
% pytest ../tut4_pymtl/regincr/test/RegIncr2stage_test.py -k test_small -s
```

```
#=========================================================================
# Regincr2stage_test
#=========================================================================

from pymtl3 import *
from pymtl3.stdlib.test_utils import run_test_vector_sim
from ..RegIncr2stage import RegIncr2stage

#-------------------------------------------------------------------------
# test_small
#-------------------------------------------------------------------------

def test_small( cmdline_opts ):
  run_test_vector_sim( RegIncr2stage(), [
    ('in_   out*'),
    [ 0x00, '?'  ],
    [ 0x03, '?'  ],
    [ 0x06, 0x02 ],
    [ 0x00, 0x05 ],
    [ 0x00, 0x08 ],
  ], cmdline_opts )

#-------------------------------------------------------------------------
# test_large
#-------------------------------------------------------------------------

def test_large( cmdline_opts ):
  run_test_vector_sim( RegIncr2stage(), [
    ('in_   out*'),
    [ 0xa0, '?'  ],
    [ 0xb3, '?'  ],
    [ 0xc6, 0xa2 ],
    [ 0x00, 0xb5 ],
    [ 0x00, 0xc8 ],
  ], cmdline_opts )

#-------------------------------------------------------------------------
# test_overflow
#-------------------------------------------------------------------------

def test_overflow( cmdline_opts ):
  run_test_vector_sim( RegIncr2stage(), [
    ('in_   out*'),
    [ 0x00, '?'  ],
    [ 0xfe, '?'  ],
    [ 0xff, 0x02 ],
    [ 0x00, 0x00 ],
    [ 0x00, 0x01 ],
  ], cmdline_opts )

#-------------------------------------------------------------------------
# test_random
#-------------------------------------------------------------------------

import random

def test_random( cmdline_opts ):

  test_vector_table = [( 'in_', 'out*' )]
  last_result_0 = '?'
  last_result_1 = '?'
  for i in range(20):
    rand_value = Bits8( random.randint(0,0xff) )
    test_vector_table.append( [ rand_value, last_result_1 ] )
    last_result_1 = last_result_0
    last_result_0 = Bits8( rand_value + 2, trunc_int=True )

  run_test_vector_sim( RegIncr2stage(), test_vector_table, cmdline_opts )
```

Figure 20: Unit Test Script for Two-Stage Registered Incrementer – A unit test for the two-stage registered incrementer shown in Figure 19 that uses test vectors and the `py.test` unit testing framework.

```
          reg_incr_0  reg_incr_1
          ----------- -----------
cycle in   in reg out in reg  out out
-------------------------------------
   ...
   3: 00 (00 (00) 01|01 (00) 01) 01
   4: 03 (03 (00) 01|01 (01) 02) 02
   5: 06 (06 (03) 04|04 (01) 02) 02
   6: 00 (00 (06) 07|07 (04) 05) 05
   7: 00 (00 (00) 01|01 (07) 08) 08
   ...
```

Figure 21: Line Trace Output for Two-Stage Registered Incrementer – This line trace is for the `test_small` test case and is annotated to show what each column corresponds to in the model. The data flow for the input value `0x03` is highlighted.

The line trace should look similar to what is shown in Figure 21. The line trace in the figure has been annotated to show what each column corresponds to in the model. If you look closely, you can see the input data propagating through both stages of the two-stage registered incrementer. Remember you can generate waveforms for all of the test cases in our new test script as follows:

```
% cd ${TUTROOT}/build
% pytest ../tut4_pymtl/regincr/RegIncr2stage_test.py --dump-vcd
% ls *.vcd
```

★ *To-Do On Your Own:* Create a three-stage registered incrementer similar in spirit to the two-stage registered incrementer in Figure 18. Verify your design by writing a test script that uses test vectors.

4.10. Parameterizing a Component with "Static" Elaboration

To facilitate component reuse and productive design-space exploration, we often want to implement parameterized components. Parameterized components take one or more parameters as constructor arguments, and then use these parameters when declaring the component's interface, defining the component's behavior in concurrent blocks, and/or structurally composing child components. A common example is to parameterize components by the bitwidth of various input and output ports. The registered incrementer in Figure 8 is designed for only eight-bit input values, but we may want to reuse this model in a different context with four-bit input values or 16-bit input values. To parameterize the port bitwidth for the registered incrementer shown in Figure 8, we add another constructor argument (which by convention we usually name `nbits`), and then we replace references to the constant `8` with a reference to `nbits`. Now we can specify the port bitwidth for our register incrementer when we construct the model. The PyMTL3 framework includes a library of parameterized FL, CL, and RTL components called `pymtl3.stdlib`. You can use the PyMTL3 GitHub repository (`http://github.com/pymtl/pymtl3`) to browse what components are available in `pymtl3.stdlib.primitive`. Figure 22 shows a combinational incrementer from `stdlib` that is parameterized by both the port bitwidth and the incrementer amount.

Figure 23 shows a more involved example where we have parameterized the number of stages in the registered incrementer. The constructor on line 13 for our multi-stage registered incrementer (`RegIncrNstage`) includes an extra argument named `nstages` (with a default value of two) that specifies how many stages should be used in the registered incrementer. Line 22 uses a Python list comprehension to create a list of `RegIncr` models. Line 26 connects the `in_` port, which is part of the interface, to the `in_` port of the first registered incrementer in the chain. Lines 30–31 use a loop to connect the `out` port of each registered incrementer to the `in_` port of the next registered incrementer. Line 35 connects the `out` port of the last registered incrementer in the chain to the `out` port

```python
class Incrementer( Component ):

  def construct( s, nbits=1, amount=1 ):

    s.in_ = InPort  ( nbits )
    s.out = OutPort ( nbits )

    @update
    def comb_logic():
      s.out @= s.in_ + amount
```

Figure 22: Parameterized Incrementer – A combinational incrementer that is parameterized by both the port bitwidth and the incrementer amount.

```python
#=========================================================================
# RegIncrNstage
#=========================================================================
# Registered incrementer that is parameterized by the number of stages.

from pymtl import *
from tut4_pymtl.regincr.RegIncr import RegIncr

class RegIncrNstage( Component ):

  # Constructor

  def construct( s, nstages=2 ):

    # Port-based interface

    s.in_ = InPort (8)
    s.out = OutPort(8)

    # Instantiate the registered incrementers

    s.reg_incrs = [ RegIncr() for _ in range(nstages) ]

    # Connect input port to first reg_incr in chain

    s.in_ //= s.reg_incrs[0].in_

    # Connect reg_incr in chain

    for i in range( nstages - 1 ):
      s.reg_incrs[i].out //= s.reg_incrs[i+1].in_

    # Connect last reg_incr in chain to output port

    s.reg_incrs[-1].out //= s.out

  # Line Tracing

  def line_trace( s ):
    return f"{s.in_} " \
           f"({'|'.join([ str(x.out) for x in s.reg_incrs ])}) " \
           f"{s.out}"
```

Figure 23: N-Stage Registered Incrementer – A parameterized registered incrementer where the number of stages is specified as an argument to the constructor.

in the interface. This example illustrates how PyMTL3 enables powerful elaboration; we can use arbitrary Python code in a component's constructor to generate complex hardware based on the constructor arguments. In traditional hardware description languages, this process is often called static elaboration since this phase happens at compile or synthesis time. In PyMTL3, the elaboration phase happens in our simulator and test scripts at "runtime," but it is essentially the same idea. To reiterate, the Python list comprehension on line 22 and the for loop on lines 30–31 does not *model* hardware, instead this code *generates* hardware. All of the code in a PyMTL3 model's constructor that is *not* in a concurrent block is used for hardware *generation*, while the code within a concurrent block is used for hardware *modeling*. Students can use whatever Python code they want for generation, but must limit themselves to a synthesizable subset for modeling.

One challenge with highly parameterized models is that they can require more complicated verification to test all of the various parameter combinations. The `pytest` framework includes sophisticated support for parameterized testing that can simplify verifying highly parameterized models. Figure 24 shows a test script for the multi-stage registered incrementer model. Because we are using a cycle-by-cycle gray-box testing strategy, the test vectors vary depending on the number of stages. Lines 23–33 define an advanced helper function that takes as input the number of stages and a list of input values and generates the corresponding test vector table. This helper function makes use of Python's standard `deque` container for carefully tracking how to set the reference outputs based on the latency of the multi-stage registered incrementer. Notice that we also use the `trunc` argument to the `Bits` constructor when creating the reference output to ensure the proper modular arithmetic.

The test script in Figure 24 uses this helper function in combination with the `pytest.mark.parametrize` decorator to create parameterized test cases. The `pytest.mark.parametrize` decorator (notice that it is `parametrize` not `parameterize`) takes two arguments: a string containing the names of arguments for the test case function and a list of values to use for those arguments. The `pytest` framework will automatically generate a set of test cases for each set of argument values.

On lines 39–55, we use `pytest.mark.parametrize` to succinctly generate eight test cases that test both two- and three-stage registered incrementers with small, large, overflow, and random input values. We use another helper function (named `mk_test_case_table`) which is provided by the PyMTL3 framework to create a test case table. A test case table compactly represents a set of test cases. Each row corresponds to a test case, and the first column is always the name of the test case. The remaining columns correspond to the test parameters. The first row of the test case table is always a special "header string" that specifies the name of each test parameter. In this example, there are two test parameters: the number of stages (`nstages`) and the test inputs (`inputs`). Notice how we use the `sample` function from the standard Python `random` module to generate a random sequence of input values. The `mk_test_case_table` creates a data structure suitable for passing into `pytest.mark.parametrize`. For technical reasons, we need to use the `**` operator to pass this data structure into `pytest.mark.parametrize`, as shown on line 46. The test function on lines 51–55 includes a `test_params` argument that will contain the test parameters corresponding to one row of the test case table. On lines 52–53, we read these test parameters, and then on lines 54–55 we use the `run_test_vector_sim` and the `mk_test_vector_table` helper functions to actually run a test.

On lines 61–64, we use `pytest.mark.parametrize` without a test case table to succinctly generate six test cases that test our multi-stage registered incrementer with one to six stages and random input values. As mentioned above, `pytest.mark.parametrize` takes two arguments: a string containing the names of arguments for the test case function (i.e., `"n"`) and a list of values to use for those arguments (i.e., `[1,2,3,4,5,6]`). The `pytest` framework generates a separate test case for each value of `n` and calls the `test_random` function with that value of `n`. Our `mk_test_vector_table` helper function enables us to make test vector tables from random input values for any number of stages.

```
#=========================================================================
# RegincrNstage_test
#=========================================================================

import collections
import pytest

from random import sample, seed

from pymtl3 import *

from pymtl3.stdlib.test_utils import run_test_vector_sim, mk_test_case_table
from ..RegIncrNstage import RegIncrNstage

# To ensure reproducible testing

seed(0xdeadbeef)

#-------------------------------------------------------------------------
# mk_test_vector_table
#-------------------------------------------------------------------------

def mk_test_vector_table( nstages, inputs ):

  inputs.extend( [0]*nstages )

  test_vector_table = [ ('in_ out*') ]
  last_results = collections.deque( ['?']*nstages )
  for input_ in inputs:
    test_vector_table.append( [ input_, last_results.popleft() ] )
    last_results.append( Bits8( input_ + nstages, trunc_int=True ) )

  return test_vector_table

#-------------------------------------------------------------------------
# Parameterized Testing with Test Case Table
#-------------------------------------------------------------------------

test_case_table = mk_test_case_table([
  (                     "nstages inputs                 "),
  [ "2stage_small",    2,       [ 0x00, 0x03, 0x06 ]   ],
  [ "2stage_large",    2,       [ 0xa0, 0xb3, 0xc6 ]   ],
  [ "2stage_overflow", 2,       [ 0x00, 0xfe, 0xff ]   ],
  [ "2stage_random",   2,       sample(range(0xff),20) ],
  [ "3stage_small",    3,       [ 0x00, 0x03, 0x06 ]   ],
  [ "3stage_large",    3,       [ 0xa0, 0xb3, 0xc6 ]   ],
  [ "3stage_overflow", 3,       [ 0x00, 0xfe, 0xff ]   ],
  [ "3stage_random",   3,       sample(range(0xff),20) ],
])
@pytest.mark.parametrize( **test_case_table )
def test( test_params, cmdline_opts ):
  nstages = test_params.nstages
  inputs  = test_params.inputs
  run_test_vector_sim( RegIncrNstage( nstages ),
    mk_test_vector_table( nstages, inputs ), cmdline_opts )

#-------------------------------------------------------------------------
# Parameterized Testing of With nstages = [ 1, 2, 3, 4, 5, 6 ]
#-------------------------------------------------------------------------

@pytest.mark.parametrize( "n", [ 1, 2, 3, 4, 5, 6 ] )
def test_random( n, cmdline_opts ):
  run_test_vector_sim( RegIncrNstage( p_nstages=n ),
    mk_test_vector_table( n, sample(range(0xff),20) ), cmdline_opts )
```

Figure 24: Unit Test Script for Parameterized Registered Incrementer – A unit test for the parameterized registered incrementer shown in Figure 23.

```
========================== test session starts ===========================
...
collected 14 items

../tut4_pymtl/regincr/test/RegIncrNstage_test.py::test[2stage_small] PASSED
../tut4_pymtl/regincr/test/RegIncrNstage_test.py::test[2stage_large] PASSED
../tut4_pymtl/regincr/test/RegIncrNstage_test.py::test[2stage_overflow] PASSED
../tut4_pymtl/regincr/test/RegIncrNstage_test.py::test[2stage_random] PASSED
../tut4_pymtl/regincr/test/RegIncrNstage_test.py::test[3stage_small] PASSED
../tut4_pymtl/regincr/test/RegIncrNstage_test.py::test[3stage_large] PASSED
../tut4_pymtl/regincr/test/RegIncrNstage_test.py::test[3stage_overflow] PASSED
../tut4_pymtl/regincr/test/RegIncrNstage_test.py::test[3stage_random] PASSED
../tut4_pymtl/regincr/test/RegIncrNstage_test.py::test_random[1] PASSED
../tut4_pymtl/regincr/test/RegIncrNstage_test.py::test_random[2] PASSED
../tut4_pymtl/regincr/test/RegIncrNstage_test.py::test_random[3] PASSED
../tut4_pymtl/regincr/test/RegIncrNstage_test.py::test_random[4] PASSED
../tut4_pymtl/regincr/test/RegIncrNstage_test.py::test_random[5] PASSED
../tut4_pymtl/regincr/test/RegIncrNstage_test.py::test_random[6] PASSED

======================= 14 passed in 0.17 seconds ========================
```

Figure 25: `pytest` **Parameterized Output –** Each line corresponds to one test case. Test cases generated using `pytest.mark.parametrize` use square brackets to denote each generated test case.

Edit the PyMTL3 source file named `RegIncrNstage.py`. Add the code on lines 28–31 from Figure 23 to connect the stages together. Then run all of the test scripts as well as a subset of the test cases as follows:

```
% cd ${TUTROOT}/build
% pytest ../tut4_pymtl/regincr/test/RegIncrNstage_test.py -v
```

The output should look similar to what is shown in Figure 25. Notice how the `pytest` framework names the generated test cases. When using a test case table, the `pytest` framework puts the test case name in square brackets after the test function name (e.g., `test[2stage_small]`). When not using a test case table, the `pytest` framework uses the arguments to the test function in square brackets after the test function name (e.g., `test_random[2]`).

As before, you can use the `-k`, `-s`, and `--dump-vcd` command line options to `pytest` to run a subset of the test cases, display a line trace, and generate waveforms. For example, the following command will run just the tests for the three-stage registered incrementer and also display a line trace.

```
% cd ${TUTROOT}/build
% pytest ../tut4_pymtl/regincr/test/RegIncrNstage_test.py -k 3stage -sv
```

★ *To-Do On Your Own:* Parameterize the input/output port bitwidth for the basic registered incrementer in Figure 8. Set the default bitwidth to be eight so that the rest of our code will still function correctly. Create a new test script named `RegIncr_param_test.py` that uses `pytest.mark.parameterize` to test various bitwidths on random input values.

4.11. Packaging a Collection of Models

We group related models into a single subdirectory (sometimes called a "subproject") within a PyMTL3 project. Packaging is the process of making a subproject available for other subprojects to use via the standard Python `import` command. Packaging simply involves adding a standard Python package configuration script named `__init__.py` to the subproject. This script is responsible for importing models within the package so as to create the package namespace. Note that if there are several nested subdirectories within the PyMTL3 project, then each of these subdirectories must have a package configuration script even if that script is empty. For example, there is a `__init__.py` file in the `tut4_pymtl` subdirectory.

Figure 26 shows a package configuration script for our `regincr` package. This script simply imports each model into the package namespace, but it is possible to also import helper functions or other classes into the package namespace.

Figure 27 shows an example session in the Python interpreter that illustrates how to import components from the `regincr` package and then use the `DefaultPassGroup` to perform a single-cycle simulation. Type these commands into the Python interpreter and observe the output.

Now try a similar interpreter session, but start the interpreter in the `build` directory. Python will report an error that it cannot find a module named `tut4_pymtl.regincr`. Python uses a special environment variable named `PYTHONPATH` to determine where to look for packages. By default the current directory is in the `PYTHONPATH` which is why our initial interpreter session is able to find the `regincr` package. Figure 28 shows how we can set the `PYTHONPATH` to the root of our project before starting the interpreter. Type these commands into the Python interpreter and observe the output.

As an aside, Figure 28 also illustrates how the PyMTL3 framework provides APIs for inspecting elaborated components. The `get_input_value_ports` method will return a list of input/output ports for an elaborated component. There are similar methods for inspecting a component's wires, child components, connections, and concurrent blocks. Thisinterface is often used when implementing new PyMTL3 tools, but can also be potentially useful when implementing highly parameterized components.

```
#=========================================================================
# regincr
#=========================================================================

from tut4_pymtl.regincr.RegIncr       import RegIncr
from tut4_pymtl.regincr.RegIncr2stage import RegIncr2stage
from tut4_pymtl.regincr.RegIncrNstage import RegIncrNstage
```

Figure 26: Configuration Script for `regincr` Package – A package configuration script is named `__init__.py` and placed in the subproject directory. The script is responsible for importing models within the package so as to create the package namespace.

```
% cd ${TUTROOT}
% python
>>> from pymtl3 import *
>>> from tut3_pymtl.regincr import RegIncr
>>> model = RegIncr()
>>> model.apply( DefaultPassGroup() )
>>> model.sim_reset()
>>> model.in_ @= 0x24
>>> model.sim_tick()
>>> model.out
Bits8(0x25)
```

Figure 27: Importing a PyMTL Package from the Tutorial Root Directory

```
% cd ${TUTROOT}/build
% env PYTHONPATH=".." python
>>> from tut3_pymtl.regincr import RegIncr
>>> model = RegIncr()
>>> model.elaborate()
>>> [ x.get_field_name() for x in model.get_input_value_ports() ]
['clk', 'in_', 'reset']
>>> [ x.get_field_name() for x in model.get_wires() ]
['reg_out']
```

Figure 28: Importing the Package from the Build Directory

5. Sort Unit

The previous section introduced the key PyMTL3 concepts and primitives that we will use to implement more complex FL, CL, and RTL models including: using the `Component` base class to define PyMTL3 models; declaring the port-based interfaces using the `InPort` and `OutPort` classes; declaring internal wires using the `Wire` class; declaring `update_ff` concurrent blocks to model logic that executes on every rising clock edge; declaring `update` concurrent blocks to model combinational logic that executes one or more times within a clock cycle; using structural composition to connect child components; and creating parameterized components. In addition, the previous section also introduced how to visualize designs with line tracing and waveforms, and how to verify designs with unit testing. In this section, we will apply what we have learned to incrementally refine a simple sort unit from an initial FL model, to a CL model, and finally an RTL model. We will also learn how to use a simulator to evaluate a design, and how to use the PyMTL3 translation tool to generate Verilog from an RTL model. Most of the code for this section is provided for you in the `tut4_pymtl/sort` subdirectory.

5.1. FL Model of Sort Unit

We begin by designing an FL model of our target sort unit. Recall that FL models implement the *functionality* but not the timing of the hardware target. Figure 29 illustrates the FL model using a cloud diagram where the "clouds" abstractly represent how logic interacts with ports and child models. Our sort unit will have four input ports for the values we want to sort and four output ports for the sorted values; all ports should used parameterized bitwidths. The sort unit should sort the values on the `in_` ports such that `out[0]` has the smallest value, `out[1]` has the second smallest value, and so on. Input/output valid bits indicate when the input/output values are valid.

Figure 30 shows how to implement an FL model for the sort unit in PyMTL3. We first create a truly functional `sort_fl` function that sorts a list of elements for our FL model to call. On lines 19 and 22, we use Python list comprehensions to create lists of four input and output ports. On lines 34 and 38, we use the standard Python `map` function to easily convert all input/output values into strings for line tracing. Notice how our line tracing code checks the input/output valid bit, and if the input/output is invalid then we clear the corresponding string to all spaces. This means the line trace will show spaces when the input/output values are invalid, but the line trace is still always a fixed width to ensure the columns stay aligned. We generally use this idea of displaying spaces in the line trace when "nothing is happening"; this makes it easy to see true activity in the line trace.

The `update_ff` concurrent block on lines 24–28 defines the actual functional-level behavior. There are many kinds of FL models, and here we create an FL model by using RTL interfaces but "magic" sorting. The `update` concurrent block in our sort unit FL model uses the `sort_fl` function and then

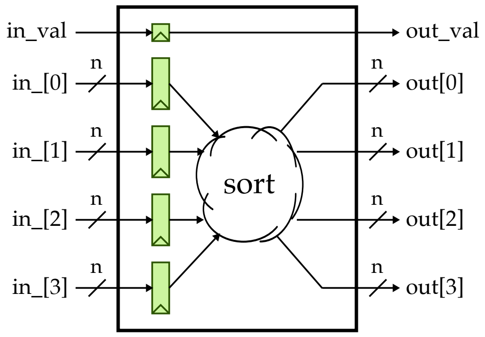


Figure 29: Cloud Diagram for Sort Unit FL Model – Cloud diagrams use "clouds" to abstractly represent logic without worry about the actual implementation details. The sort unit FL model takes four input values and sorts them such that the `out[0]` port has the smallest value and the `out[3]` port has the largest value. Input/output valid bits indicate when the input/output values are valid.

```
#=========================================================================
# Sort Unit FL Model
#=========================================================================
# Models the functional behavior of the target hardware but not the
# timing.

from copy import deepcopy
from pymtl3 import *

class SortUnitFL( Component ):

  # Constructor

  def construct( s, p_nbits=8 ):

    s.in_val = InPort ()
    s.in_    = [ InPort (p_nbits) for _ in range(4) ]

    s.out_val = OutPort()
    s.out     = [ OutPort(p_nbits) for _ in range(4) ]

    @update_ff
    def block():
      s.out_val <<= s.in_val
      for i, v in enumerate( sorted( s.in_ ) ):
        s.out[i] <<= v

  # Line tracing

  def line_trace( s ):

    in_str = '{' + ','.join(map(str,s.in_)) + '}'
    if not s.in_val:
      in_str = ' '*len(in_str)

    out_str = '{' + ','.join(map(str,s.out)) + '}'
    if not s.out_val:
      out_str = ' '*len(out_str)

    return f"{in_str}|{out_str}"
```

Figure 30: Sort Unit FL Model – FL model of four-element sort unit corresponding to Figure 29.

uses a loop to write the sorted values to the output ports. The valid bit from the `in_val` port is written directly to the `out_val` port. Note that although using `update_ff` blocks here works for simulation, the Verilog translation pass will see `sort_fl` as not translatable.

Notice that although this model in no way attempts to capture any timing of the hardware target, it is still a "single-cycle" model. This is due to the PyMTL3 semantics of `update_ff` concurrent blocks and non-blocking assignments, and this is why we show input registers in the cloud diagram in Figure 29. Although it is also possible to implement FL models using `update` concurrent blocks, we have found using `update_ff` concurrent blocks to be significantly easier. Using `update` concurrent blocks means the block can be called multiple times in a cycle, increases the likelihood of creating combinational loops when composing FL models, and complicates incrementally refining an FL model into a CL model.

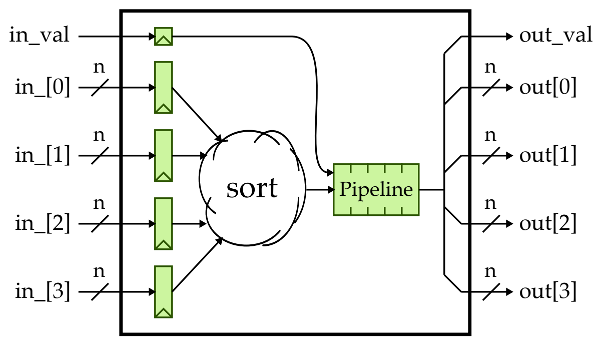


Figure 31: Cloud Diagram for Sort Unit CL Model – The CL model completely sorts the input values in the first cycle, and then uses a pipeline object to model the pipeline latency.

We do not explicitly handle resetting the valid bit, but we instead rely on the PyMTL3 framework, which guarantees that signals are reset to zero by default. Leveraging this guarantee simplifies our FL (and CL) models, but keep in mind that RTL models must still explicitly handle resetting state.

The PyMTL3 model is in `SortUnitFL.py` and the corresponding test script is in `SortUnitFL_test.py`. This test script first tests the `sort_fl` function, and then uses test vector tables similar in spirit to the unit testing for the registered incrementer in Figure 20. With the test-driven design principle in mind, the four test vectors that passed the tests of the `sort_fl` function can be directly reused to test the `SortUnifFL` component. The test script for `SortUnitFL` includes four directed test cases and one random test case. Note that we usually try to ensure that the very first test case is always the simplest possible test case we can imagine. For this model, our first test case simply sorts a single set of four input values. You can run all of the tests and display the line trace for the basic test case as follows:

```
% cd ${TUTROOT}/build
% pytest ../tut4_pymtl/sort/SortUnitFL_test.py -v
% pytest ../tut4_pymtl/sort/SortUnitFL_test.py -k test_basic -s
```

Once we have implemented a FL model, we can then use this model to enable early verification work. We can write and check tests using the FL model, and then gradually these same tests can be used with the CL and RTL models. Using the FL model to write tests also ensures if the CL or RTL models fail a test, it is more likely due to the CL or RTL implementation itself as opposed to an incorrect test case.

★ *To-Do On Your Own:* Add another directed test case that specifically tests for when the inputs are already sorted in increasing and then decreasing order. Add another random test case for a sort unit with 12-bit input/output values.

5.2. CL Model of Sort Unit

Once we have a reasonable FL model, we can manually refine this model into a CL model. Recall that CL models capture the *cycle-approximate behavior* of a hardware target. We can achieve this with additional logic to track the cycle-level performance of our target hardware. In this case, we will assume that our target hardware is a pipelined sort unit, although we may not know yet how many stages our final design will use. Figure 31 illustrates the CL model using a cloud diagram. The high-level approach is to completely sort the input values in the first cycle, and then to pipeline the sorted results some number of cycles to model the cycle-level performance of the target hardware.

```
 1 #=========================================================================
 2 # Sort Unit CL Model
 3 #=========================================================================
 4 # Models the cycle-approximate timing behavior of the target hardware.
 5
 6 from collections import deque
 7 from copy import deepcopy
 8
 9 from pymtl3 import *
10
11 from tut4_pymtl.sort.SortUnitFL import sort_fl
12
13 class SortUnitCL( Component ):
14
15   # Constructor
16
17   def construct( s, nbits=8, nstages=3 ):
18
19     s.in_val  = InPort ()
20     s.in_     = [ InPort (nbits) for _ in range(4) ]
21
22     s.out_val = OutPort()
23     s.out     = [ OutPort(nbits) for _ in range(4) ]
24
25     s.pipe    = deque( [[0,0,0,0,0]]*(nstages-1) )
26
27     @update_ff
28     def block():
29       s.pipe.append( deepcopy( [s.in_val] + sort_fl(s.in_) ) )
30       data = s.pipe.popleft()
31       s.out_val <<= data[0]
32       for i, v in enumerate( data[1:] ):
33         s.out[i] <<= v
34
35   # Line tracing
36
37   def line_trace( s ):
38
39     in_str = '{' + ','.join(map(str,s.in_)) + '}'
40     if not s.in_val:
41       in_str = ' '*len(in_str)
42
43     out_str = '{' + ','.join(map(str,s.out)) + '}'
44     if not s.out_val:
45       out_str = ' '*len(out_str)
46
47     return "{}|{}".format( in_str, out_str )
```

Figure 32: Sort Unit CL Model – CL model of four-element sort unit corresponding to Figure 31.

Figure 32 shows how to implement a CL model for the sort unit in PyMTL3. On line 23, we instantiate a `deque` object (i.e., a doubly ended queue) from the standard Python `collections` module. The `deque` will be used to model the pipeline latency: each cycle we will append a value to the back of the `deque` and pop a value from the front of the `deque`. Depending on how we initialize the `deque`, it

```
cycle input ports   output ports
-------------------------------------
   3:              |
   4: {04,02,03,01}|
   5:              |
   6:              |
   7:              |{01,02,03,04}
   8:              |
```

Figure 33: Line Trace Output for Sort Unit CL Model – This line trace is for the test_basic test case and is annotated to show what each column corresponds to in the model.

will take some number of cycles for a value to propagate from the back to the front of the deque and this latency corresponds to the pipeline latency.

The update_ff concurrent block on lines 27–33 defines the actual cycle-level behavior. We first sort the input values using the sort_fl function and append the corresponding sorted list of four values along with the input valid bit to the back of the deque (line 29). We then pop the next list of four values from the front of the deque (line 30), write the valid bit to the out_val port (line 31), and write the sorted list to the out ports (lines 32–33). Notice how line 25 initializes the deque to contain nstages-1 entries (each entry is list of four values). If nstages is three, then there are initially two entries in the deque. Every cycle we will append a value to the back of the deque and pop a value from the front of the deque. So it will take three cycles for a value to propagate from the back to the front of the deque. We initialize the deque to contain nstages-1 instead of nstages elements, because we have carefully designed our model to cleanly support the case when nstages is one. In this case the deque is initially empty. On line 29 we will append the list of sorted values to the deque, and on line 30 we will immediately pop this same list of sorted values from the deque. In this case, the update_ff concurrent block itself gives us a single-cycle delay, and the deque does not add any additional latency.

A key point to note is the use of the deepcopy function from the standard Python copy module on line 29. Recall that simply assigning one Python name to another name does *not* create a copy, but results in two names referring to the same object. Without this deepcopy, the list we append to the back of the deque contains references to Bits objects that are also referenced elsewhere in the framework. The deepcopy function appends a copy of the input valid bit and sorted list to the deque. Copying objects is often necessary when reading values from an input port and storing these values in a standard Python data structure. If your FL or CL model is exhibiting strange behavior where signals seem not to change or change to arbitrary values, you may want to carefully consider whether or not you are forgetting to copy objects.

The PyMTL3 model is in SortUnitCL.py and the corresponding test script is in SortUnitCL_test.py. This test script uses parameterized testing similar in spirit to the unit testing for the parameterized registered incrementer in Figure 24. The test script generates 18 test cases for directed and random testing of the sort unit CL model with different input values and numbers of stages. Take a closer look at this test script before continuing. You can run all of the tests and display the line trace for one of the three-stage test cases as follows:

```
% cd ${TUTROOT}/build
% pytest ../tut4_pymtl/sort/test/SortUnitCL_test.py -v
% pytest ../tut4_pymtl/sort/test/SortUnitCL_test.py -k 3stage_stream -s
```

Figure 33 shows the line trace for the basic test case. Study the line trace to see how the CL model captures the cycle-level performance of our sort unit. Imagine we want to integrate this sort unit into a larger system. Because our sort unit CL model is parameterized by the number of stages, it would be relatively simple to explore how the sort unit latency impacts the overall system-level performance. This initial design-space exploration can enable a designer to determine a reasonable target latency for the sort unit without the need for tediously implementing many different RTL models, each with different pipeline latencies. Once we have implemented an RTL model with a specific pipeline latency, we might still want to use the CL model as part of our overall system-level model, since its simplicity leads to much higher simulator performance.

★ *To-Do On Your Own:* Experiment with what happens if you initialize the `deque` to have just `nstage` instead of `nstages-1` elements. Experiment with removing the `deepcopy`. Generate waveforms for one of the test cases and confirm that signals are recorded in the waveform (e.g., `in_` and `out` ports) but not arbitrary Python data structures used within a model (e.g., the `deque`).

5.3. Flat RTL Model of Sort Unit

Let's assume we used our sort unit CL model to explore the cycle-level performance of our system, and we have settled on implementing a three-stage pipelined sort unit. We now manually refine this model into an RTL model. Recall that RTL models are *cycle-accurate*, *resource-accurate*, and *bit-accurate* representations of hardware. Although RTL models are usually the most tedious to construct, they are also the most accurate with respect to the target hardware. Note that this is an iterative process: our CL design-space exploration might suggest a target three-stage pipeline, but then our RTL design-space exploration might reveal that a two-stage pipeline is much more efficient in terms of area, energy, or timing. Based on these RTL insights we can revisit our CL model and analyze the system-level impact of using a two-stage pipeline latency. Figure 34 illustrates the RTL model using a block diagram. Each min/max unit compares its inputs and sends the smaller value to the top output port and the larger value to the bottom output. This specific implementation is pipelined into three stages, such that the critical path should be through a single min/max unit. Input and output valid signals indicate when the input and output elements are valid. We are essentially implementing a pipelined bitonic sorting network.

Notice that we register the inputs but we do not register the outputs. In other words, we register the inputs as soon as possible, but there is almost a full cycle's worth of work before the outputs are stable. When working with larger blocks we usually need to decide whether to use registered inputs or registered outputs, and it is important that we adopt a uniform policy. When some blocks use registered inputs and others use registered outputs, composing them can create either long critical paths or "dead cycles" where very little work happens beyond simply transferring data. In this course, we will adopt the general policy of using registered inputs for larger blocks. As long as all modules roughly adhere to this policy then we can focus on the critical path of each larger module in isolation and be confident that composing these blocks should not cause significant timing issues.

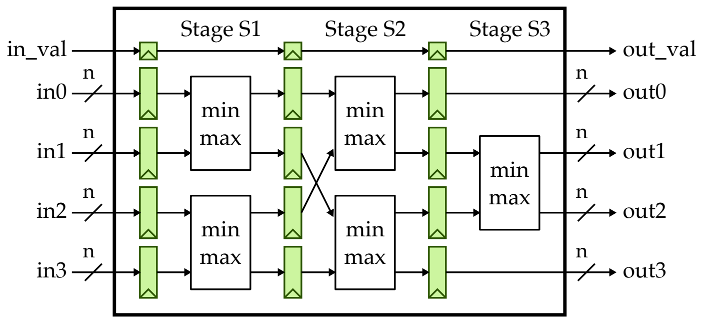


Figure 34: Block Diagram for Sort Unit RTL Model – The RTL model implements a three-stage pipelined, bitonic sorting network.

```
#=========================================================================
# SortUnitFlatRTL
#=========================================================================

from pymtl3 import *

class SortUnitFlatRTL( Component ):

  def construct( s, nbits=8 ):

    #---------------------------------------------------------------------
    # Interface
    #---------------------------------------------------------------------

    s.in_val  = InPort ()
    s.in_     = [ InPort (nbits) for _ in range(4) ]

    s.out_val = OutPort()
    s.out     = [ OutPort(nbits) for _ in range(4) ]

    #---------------------------------------------------------------------
    # Stage S0->S1 pipeline registers
    #---------------------------------------------------------------------

    s.val_S1 = Wire()
    s.elm_S1 = [ Wire(nbits) for _ in range(4) ]

    @update_ff
    def pipereg_S0S1():

      if s.reset:
        s.val_S1 <<= 0
      else:
        s.val_S1 <<= s.in_val

      for i in range(4):
        s.elm_S1[i] <<= s.in_[i]

    #---------------------------------------------------------------------
    # Stage S1 combinational logic
    #---------------------------------------------------------------------

    s.elm_next_S1 = [ Wire(nbits) for _ in range(4) ]

    @update
    def stage_S1():

      # Sort elements 0 and 1

      if s.elm_S1[0] <= s.elm_S1[1]:
        s.elm_next_S1[0] @= s.elm_S1[0]
        s.elm_next_S1[1] @= s.elm_S1[1]
      else:
        s.elm_next_S1[0] @= s.elm_S1[1]
        s.elm_next_S1[1] @= s.elm_S1[0]

      # Sort elements 2 and 3

      if s.elm_S1[2] <= s.elm_S1[3]:
        s.elm_next_S1[2] @= s.elm_S1[2]
        s.elm_next_S1[3] @= s.elm_S1[3]
      else:
        s.elm_next_S1[2] @= s.elm_S1[3]
        s.elm_next_S1[3] @= s.elm_S1[2]

    ...
```

Figure 35: Sort Unit Flat RTL Model – RTL model of four-element sort unit corresponding to Figure 34. For simplicity only the interface and first pipeline stage are shown.

```
cycle input ports     stage S1      stage S2      stage S3    output ports
-----------------------------------------------------------------------
   1r             |             |             |             |
   2r             |             |             |             |
   3:             |             |             |             |
   4: {04,02,03,01}|             |             |             |
   5:             |{04,02,03,01}|             |             |
   6:             |             |{02,04,01,03}|             |
   7:             |             |             |{01,03,02,04}|{01,02,03,04}
   8:             |             |             |             |
```

Figure 36: Line Trace Output for Sort Unit RTL Model – This line trace is for the `test_basic` test case and is annotated to show what each column corresponds to in the model. If the valid bit is not set, then the corresponding list of values is not shown.

Figure 35 shows how to implement a flat RTL model for the sort unit in PyMTL3. We say this model is "flat" because it does not instantiate any additional child models. For simplicity, only the first pipeline stage of the sort unit RTL model is shown. We cleanly separate the sequential logic (modeled with `update_ff` concurrent blocks) from the combinational logic (modeled with `update` concurrent blocks). We use comments and explicit suffixes to make it clear what pipeline stage we are modeling.

Since RTL models are meant to model real hardware, we cannot rely on the PyMTL3 framework to reset state. Line 31 uses the implicit `s.reset` signal to reset the valid bit register to zero in the first stage of the pipeline. Simple loops with bounds fixed at elaboration are allowed within RTL models. Lines 36–37 illustrate a loop that iterates over the `in_` ports to model the input registers. Lines 45–64 correspond to the first stage in Figure 34 with two min/max units.

The PyMTL3 model is in `SortUnitFlatRTL.py` and the corresponding test script is in `SortUnitFlatRTL_test.py`. The test script includes four directed tests and one random test. Take a closer look at this test script before continuing. You can run all of the tests and display the line trace for the basic test case as follows:

```
% cd ${TUTROOT}/build
% pytest ../tut4_pymtl/sort/test/SortUnitFlatRTL_test.py -v
% pytest ../tut4_pymtl/sort/test/SortUnitFlatRTL_test.py -k test_basic -s
```

The line trace for the sort unit RTL model is shown in Figure 36. Cycle 1 and cycle 2 are during the reset phase. Cycle 3 doesn't have a valid input. On cycle 4, there is a valid set of four input values available on the input ports, and on cycle 5, we can see that this set of four values is now in the first set of pipeline registers. Recall that our line trace shows the state at the beginning of the corresponding cycle. During cycle 5, pipeline stage S1 swaps elements 0 and 1, and also swaps elements 2 and 3. We can see the result of these swaps by looking at the four values on cycle 5 at the beginning of pipeline stage S2. During cycle 6, pipeline stage S2 swaps elements 0 and 2, and also swaps elements 1 and 3. During cycle 7, pipeline stage S1 swaps elements 1 and 2 before writing the results to the output ports. Compare the cycle-level behavior of the sort unit CL model in Figure 33 and the sort unit RTL model in Figure 36. While obviously the internals of each model are very different, from the perspective of just the input/output ports these two models have the exact same cycle-level behavior. An unsorted set of four values is consumed by the sort unit model on cycle 4, and a sorted set of four values is produced by the sort unit model on cycle 7. We say that the sort unit CL model is *cycle accurate* with respect to the sort unit RTL model. Often our CL models will be *cycle approximate*, meaning they will approximately model the cycle-level behavior of the RTL model. This is the key to CL modeling; CL models should capture the CL timing behavior, but they need not accurately model the actual target hardware.

★ *To-Do On Your Own:* Make a copy of the sorter implementation file so you can put things back to the way they were when you are finished. The sorter currently sorts the four input numbers from smallest to largest. Change to the sorter implementation so it sorts the numbers from largest to smallest. Recompile and rerun the unit test and verify that the tests are no longer passing. Modify the tests so that they correctly capture the new expected behavior. You might want to make use of the optional `reverse` argument to the standard Python `sorted` function.

```
% cd ${TUTROOT}/build
% python
>>> sorted( [ 3, 1, 7, 5 ] )
[1, 3, 5, 7]
>>> sorted( [ 3, 1, 7, 5 ], reverse=True )
[7, 5, 3, 1]
```

5.4. Structural RTL Model of Sort Unit

The sort unit flat RTL model is complex and monolithic and it fails to really exploit the structure inherent in the sorter. We can use modularity and hierarchy to divide complicated designs into smaller more manageable units; these smaller units are easier to design and can be tested independently before integrating them into larger, more complicated designs.

Figure 37 shows how to implement a structural RTL model for the sort unit in PyMTL3. We say this model is "structural" because it only instantiates other child models. For simplicity, only the first pipeline stage of the sort unit RTL model is shown. Even though we are using a structural implementation strategy, we still cleanly separate the sequential child components from the combinational child components. We still use comments and explicit suffixes to make it clear what pipeline stage we are modeling.

Notice on lines 28–31 we are using register models from `stdlib`. On line 6, we import the `Reg` (simple positive-edge triggered register) and `RegRst` (positive-edge triggered register with reset) components. Notice our use of a loop to connect the `in_` ports in the interface to the `in_` ports in the `Reg` component. As shown on lines 38–46, we usually instantiate a child component, and then we use connect statements to implement structural composition. There is no need to declare intermediate wires; we can directly connect ports between two different components.

The PyMTL3 model is in `SortUnitStructRTL.py` and the corresponding test script is in `SortUnitStructRTL_test.py`. The test script includes four directed tests and one random test. Take a closer look at this test script before continuing; notice how the test script is able to import a helper function (`mk_test_vector_table`) from `SortUnitCL_test.py`. This ability to share test vectors, cases, and/or harnesses across many different test scripts is a significant benefit of the `pytest` framework.

```python
#=========================================================================
# SortUnitStructRTL
#=========================================================================

from pymtl3 import *

from pymtl3.stdlib.primitive import Reg, RegRst

from tut4_pymtl.sort.MinMaxUnit import MinMaxUnit

class SortUnitStructRTL( Component ):

  def construct( s, nbits=8 ):

    #---------------------------------------------------------------------
    # Interface
    #---------------------------------------------------------------------

    s.in_val  = InPort ()
    s.in_     = [ InPort (nbits) for _ in range(4) ]

    s.out_val = OutPort()
    s.out     = [ OutPort(nbits) for _ in range(4) ]

    #---------------------------------------------------------------------
    # Stage S0->S1 pipeline registers
    #---------------------------------------------------------------------

    s.val_S0S1 = RegRst(Bits1)
    s.val_S0S1.in_ //= s.in_val

    s.elm_S0S1 = [ Reg(mk_bits(nbits)) for i in range(4) ]

    for i in range(4):
      s.elm_S0S1[i].in_ //= s.in_[i]

    #---------------------------------------------------------------------
    # Stage S1 combinational logic
    #---------------------------------------------------------------------

    s.minmax0_S1 = m = MinMaxUnit(nbits)
    m.in0 //= s.elm_S0S1[0].out
    m.in1 //= s.elm_S0S1[1].out

    s.minmax1_S1 = m = MinMaxUnit(nbits)
    m.in0 //= s.elm_S0S1[2].out
    m.in1 //= s.elm_S0S1[3].out

    ...
```

Figure 37: Sort Unit Structural RTL Model – RTL model of four-element sort unit corresponding to Figure 34. For simplicity only the interface and first pipeline stage are shown.

★ *To-Do On Your Own:* The structural implementation is incomplete because the actual implementation of the min/max unit in `MinMaxUnit.py` is not finished. You should go ahead and implement the logic for the min/max unit, and then *as always you should write a unit test to verify the functionality of your min/max unit!* Add some line tracing for the min/max unit. You should have enough experience based on the previous sections to be able to create a unit test from scratch and run it using `pytest`. You should name the new test script `MinMaxUnit_test.py`. You can use the registered incrementer model as an example for both implementing the min/max unit and for writing the corresponding test script. Once your min/max unit is complete and tested, then test the structural sorter implementation like this:

```
% cd ${TUTROOT}/build
% pytest ../tut4_pymtl/sort/test/SortUnitStructRTL_test.py -v
% pytest ../tut4_pymtl/sort/test/SortUnitStructRTL_test.py -k test_basic -s
```

The line trace for the sort unit structural RTL model should be the same as in Figure 36, since these are really just two different implementations of the sort unit RTL.

5.5. Evaluating Sort Unit using a Simulator

So far we have focused on implementing and verifying our design, but our ultimate goal is to actually evaluate a design. We do not use unit tests for evaluation; instead we use a *simulator script* which has been designed for quantitatively measuring the cycle-level performance of a specific implementation on a given input dataset. For this tutorial, we will create a simulator to compare the various models of our sort unit when executing various input datasets.

The simulator script is in `sort-sim`. A simplified version of the `main` function in the script is shown in Figure 38. The simulator script is responsible for handling command line arguments, creating input datasets, instantiating and elaborating the design, ticking the simulator until the evaluation is finished, and reporting various statistics. Lines 8–10 create an input pattern based on the `--input` command line parameter. Simulator scripts can use standard Python to flexible generate a wide variety of different input patterns. Lines 16–20 define a standard Python dictionary that maps strings to model types. Then on line 22, we can simply use this dictionary to instantiate the correct model based on the `--impl` command line option. The simulator will take care of conditionally generating waveforms based on the `--dump-vcd` command line option by crafting a dictionary and pass to a `stdlib` test utility function. Lines 34 turn on line tracing based on the `--trace` command line option. The main simulator loop on lines 41–60 iterates through the input dataset and sets the corresponding input ports. The simulator loops keeps a counter to track how many valid outputs have been received, and thus to determine when to stop the simulation. A key difference between a simulator and a unit test, is that the simulator should also report various statistics that help us evaluate our design. The `--stats` command line option will display the number of cycles to finish processing the input dataset, and the average number of cycles per sort. You can run the simulator script for the sort unit CL and RTL models as follows:

```
% cd ${TUTROOT}/build
% ../tut4_pymtl/sort/sort-sim --stats --impl cl
% ../tut4_pymtl/sort/sort-sim --stats --impl rtl-flat
% ../tut4_pymtl/sort/sort-sim --stats --impl rtl-struct
```

```python
opts = parse_cmdline()

# Create input datasets

ninputs = 100
inputs  = []

if opts.input == "random":
  for i in range(ninputs):
    inputs.append( [ randint(0,0xff) for _ in range(4) ] )

...

# Instantiate and elaborate the design

model_impl_dict = {
  'rtl-flat'   : SortUnitFlatRTL,
  'rtl-struct' : SortUnitStructRTL,
}

model = model_impl_dict[ opts.impl ]()

...

unique_name = f"sort-{opts.impl}-{opts.input}"

cmdline_opts = {
  'dump_vcd': f"{unique_name}" if opts.dump_vcd else '',
  'test_verilog': 'zeros' if opts.translate else '',
}

model = config_model_with_cmdline_opts( model, cmdline_opts, duts=[] )
model.apply( DefaultPassGroup( linetrace=opts.trace ) )

model.sim_reset()

# Tick simulator until evaluation is finished

counter = 0
while counter < ninputs:

  if model.out_val:
    counter += 1

  if inputs:
    model.in_val @= 1
    for i,v in enumerate( inputs.pop() ):
      model.in_[i] @= v

  else:
    model.in_val @= 0
    for i in range(4):
      model.in_[i] @= 0

  model.sim_eval_combinational()
  if opts.trace:
    model.print_line_trace()

  model.sim_tick()

# Report various statistics

if opts.stats:
  print( "num_cycles          = {}".format( sim.ncycles ) )
  print( "num_cycles_per_sort = {:1.2f}".format( sim.ncycles/(1.0*ninputs) ) )
```

Figure 38: Simplified Simulator Script for Sort Unit – The simulator script is responsible for handling command line arguments, creating input datasets, instantiating and elaborating the design, ticking the simulator until the evaluation is finished, and reporting various statistics.

Not surprisingly, it should take one cycle on average since our CL model captures the timing behavior of a fully pipelined implementation, and our RTL models actually implement a fully pipelined design. The number of cycles per sort is slightly greater than one due to pipeline startup overhead.

You can experiment with other input datasets like this:

```
% cd ${TUTROOT}/build
% ../tut4_pymtl/sort/sort-sim --stats --impl cl --input random
% ../tut4_pymtl/sort/sort-sim --stats --impl cl --input sorted-fwd
% ../tut4_pymtl/sort/sort-sim --stats --impl cl --input sorted-rev
```

You can display a line trace and generate waveforms like this:

```
% cd ${TUTROOT}/build
% ../tut4_pymtl/sort/sort-sim --stats --impl rtl-struct --trace --dump-vcd
```

Note that the simulator does absolutely no verification! If you have not actually completed the real implementation of the min/max unit, the `rtl-struct` implementation will still run and actually the simulator will report what looks to be reasonable performance results; *even though the structural implementation is not at all functionally correct*. The take-away here is that you should not use a simulator script for verification; your testing strategy should be comprehensive enough that once you get to the evaluation you are confident that your design is fully functional.

★ *To-Do On Your Own:* Add a fourth random input dataset where all of the input values are less than 16. Add a new choice to the `--input` command line option corresponding to this new input dataset. Use the simulator and line tracing to experiment with this new dataset on various implementations of the sort unit.

5.6. Translating RTL Model of Sort Unit to Verilog

After we have refined our design from an initial FL model, to a CL model, and to an RTL model; rigorously verified our design using unit testing; and evaluated our design using a simulator; we are finally ready to translate the RTL model into an industry standard HDL. The generated HDL can be used to verify that our RTL model is indeed synthesizable, create faster simulators, drive an FPGA toolflow for emulation and/or prototyping, or drive an ASIC toolflow for accurately estimating area, energy, and timing. PyMTL3 currently supports translating an RTL model into SystemVerilog and Verilog. The framework's use of a clean DSL/passes split can enable adding translation passes for other HDLs in the future.

Figure 39 shows an example session in the Python interpreter that illustrates how to use the `VerilogTranslationPass` from the PyMTL3 framework to translate an RTL model into Verilog. Type these commands into the Python interpreter and observe the output. Then browse the files generated during translation. Browse `SortUnitFlatRTL__nbits_8__pickled.v` to see the Verilog generated by the translation tool. The suffix `__nbits_8__pickled` corresponds to a mangled string containing parameter names and values and tells the user that all components involved are pickled into a single file; this ensures that the generated Verilog module name is unique across different instantiations of the same parameterized model. Notice that the translation pass preserves the model hierarchy, unrolls lists, and uses readable name mangling from PyMTL3 to Verilog names. `update_ff` concurrent blocks are translated into `always_ff` concurrent blocks, and `update` concurrent blocks are translated into `always_comb` concurrent blocks. Also notice that for each concurrent block, the translation pass in-

```
% cd ${TUTROOT}/build
% env PYTHONPATH=".." python
>>> from pymtl3 import *
>>> from pymtl3.passes.backends.verilog import *
>>> from tut3_pymtl.sort import SortUnitFlatRTL
>>> model = SortUnitFlatRTL()
>>> model.set_metadata( VerilogTranslationPass.enable, True )
>>> model.apply( VerilogTranslationPass() )
% ls
SortUnitFlatRTL__nbits_8__pickled.v
```

Figure 39: Translating an RTL Model into Verilog – The `VerilogTranslationPass` translates an RTL model into Verilog. As a quick aside, the `VerilogTranslationImportPass` will use the Verilator tool and various generated wrappers to creates a new PyMTL3 model that internally contains its own cycle-accurate simulator for the translated Verilog.

cludes the corresponding PyMTL3 code as a comment directly above the generated Verilog. This can be useful when debugging incorrect translations.

Note that the translation pass only generates Verilog source code. We also have the `VerilogTranslationImportPass` that: 1) translate an RTL model into Verilog; (2) use the open-source Verilator tool to translate the Verilog into C++; (3) generate a C++ wrapper; (4) compile this wrapper and the C++ generated by Verilator into a shared library; and (5) generate a PyMTL3 wrapper around the shared library. Essentially, this means the translation-import flow creates a new PyMTL3 model that internally contains its own cycle-accurate simulator for the translated Verilog. As part of this process, the translation tool generates several extra files in the build directory. Feel free to browse through the C++ and PyMTL3 wrappers. This powerful feature enables us to seamlessly use the exact same test scripts to verify the functionality of the translated Verilog.

Figure 40 shows a unit test with support for testing a translated model. This test is very similar to the initial test script for the registered incrementer shown in Figure 13, except of course our sort unit requires many more input and output values. The `cmdline_opts` arguments are handled specially by the pytest plugin inside PyMTL3 framework. It creates a dictionary and holds commandline parameter pairs. For example, `cmdline_opts[ťest_verilog]` is set toTrue when the `--test-verilog` command line option is given to `pytest`. On line 17, we use the `config_model_with_cmdline_opts` API to transform the model using command line options. The API returns the changed model. Both `model` and `duts` parameters are required if you want to simulate the top-level component. We can now test both the PyMTL3 RTL and the translated Verilog as follows:

```
% cd ${TUTROOT}/build
% pytest ../tut4_pymtl/sort/test/SortUnitFlatRTL_v_test.py --dump-vcd
% mv tut4_pymtl.sort.test.SortUnitFlatRTL_v_test__test_verilate.vcd sort-pymtl.vcd
% pytest ../tut4_pymtl/sort/test/SortUnitFlatRTL_v_test.py --dump-vcd --test-verilog
% ls *SortUnitFlatRTL_v_test.*.vcd
```

We save the generated VCD file from the first `pytest` run as `sort-pymtl.vcd`. When testing with the `--test-verilog` command line option during the second `pytest` run, the PyMTL3 framework will generate two different VCD files (with relatively long file names). One file corresponds to the PyMTL3 wrapper, and the other file corresponds to the actual Verilog design. Browse all three generated waveforms to understand the difference.

```
#=========================================================================
# SortUnitFlatRTL_v_test
#=========================================================================

from pymtl3 import *
from pymtl3.passes.backends.verilog import *
from pymtl3.stdlib.test_utils import config_model_with_cmdline_opts

from tut4_pymtl.sort.SortUnitFlatRTL import SortUnitFlatRTL

def test_verilate( cmdline_opts ):

  # Configure the model with dump_vcd and test_verilog flags

  model = SortUnitFlatRTL(8)

  model = config_model_with_cmdline_opts( model, cmdline_opts, duts=[] ) # use model itself

  # Create and reset simulator

  model.apply( DefaultPassGroup(linetrace=True) )
  model.sim_reset()

  # Helper function

  def t( in_val, in_, out_val, out ):

    model.in_val @= in_val
    for i,v in enumerate( in_ ):
      model.in_[i] @= v

    model.sim_eval_combinational()

    assert model.out_val == out_val
    if out_val:
      for i,v in enumerate( out ):
        assert model.out[i] == v

    model.sim_tick()

  # Cycle-by-cycle tests

  t( 0, [ 0x00, 0x00, 0x00, 0x00 ], 0, [ 0x00, 0x00, 0x00, 0x00 ] )
  t( 1, [ 0x03, 0x09, 0x04, 0x01 ], 0, [ 0x00, 0x00, 0x00, 0x00 ] )
  t( 1, [ 0x10, 0x23, 0x02, 0x41 ], 0, [ 0x00, 0x00, 0x00, 0x00 ] )
  t( 1, [ 0x02, 0x55, 0x13, 0x07 ], 0, [ 0x00, 0x00, 0x00, 0x00 ] )
  t( 0, [ 0x00, 0x00, 0x00, 0x00 ], 1, [ 0x01, 0x03, 0x04, 0x09 ] )
  t( 0, [ 0x00, 0x00, 0x00, 0x00 ], 1, [ 0x02, 0x10, 0x23, 0x41 ] )
  t( 0, [ 0x00, 0x00, 0x00, 0x00 ], 1, [ 0x02, 0x07, 0x13, 0x55 ] )
  t( 0, [ 0x00, 0x00, 0x00, 0x00 ], 0, [ 0x00, 0x00, 0x00, 0x00 ] )
  t( 0, [ 0x00, 0x00, 0x00, 0x00 ], 0, [ 0x00, 0x00, 0x00, 0x00 ] )
```

Figure 40: Unit Test Script for Sort Model with Verilog Translation – The `test_verilog` argument is handled specially by the PyMTL framework; it is set to `True` when the `--test-verilog` command line option is given to `py.test`.

You will probably notice that the second `pytest` run takes a bit longer than the first `pytest` run. This is because the second `pytest` run must go through all of the steps to translate the design to Verilog and ultimately create a new PyMTL3 model that internally contains its own cycle-accurate simulator for this translated Verilog. The Verilog translation pass caches the result of translation to reduce this overhead when testing the same model many times. If you run `pytest` again with the `--test-verilog` command line option, it will execute faster since the pass realizes it can just reuse the translated model from before.

However, sometimes the translation pass can get confused; you may need to remove all of the content in the build directory and do a "clean" build to occasionally fix issues with translation like this:

```
% cd ${TUTROOT}/build
% rm -rf *
% pytest ../tut4_pymtl/sort/test/test/SortUnitFlatRTL_v_test --dump-vcd --test-verilog
```

If you take a closer look at the `SortUnitFlatRTL_test.py` test script, you will see that the `run_test_vector_sim` helper function accepts `cmdline_opts` as an argument. This enables us to test the translated Verilog on all of our tests as follows:

```
% cd ${TUTROOT}/build
% pytest ../tut4_pymtl/sort -v --test-verilog
```

You should see that `pytest` tests the translated Verilog for the min/max unit and our sort unit RTL models, but skips testing the FL and CL models since these models cannot be translated into Verilog.

Once we have verified that our RTL models can be correctly translated into Verilog, we will ultimately use the simulator script (with the `--translate` command line option) to generate the actual Verilog that can be used to drive an FPGA or ASIC toolflow. We can at the same time also generate waveforms to drive power analysis in an ASIC toolflow. The following commands use the simulator script to generate the Verilog for the sort unit flat RTL model and three VCD files corresponding to the three input datasets.

```
% ../tut4_pymtl/sort/sort-sim --impl rtl-flat --input random     --translate --dump-vcd
% ../tut4_pymtl/sort/sort-sim --impl rtl-flat --input sorted-fwd --translate --dump-vcd
% ../tut4_pymtl/sort/sort-sim --impl rtl-flat --input sorted-rev --translate --dump-vcd
```

★ *To-Do On Your Own:* Experiment with translating the sort unit structural RTL model to Verilog. Verify that all of the test cases for the structural RTL model pass on the translated model, and use the simulator to generate the Verilog and VCD files for all three input patterns.

6. Greatest Common Divisor Unit

The previous section introduced the process of refining a design from an initial FL model, to a CL model, and finally an RTL model. In this section, we will apply what we have learned to study a more complicated hardware unit that calculates the greatest common divisor (GCD) of two input operands. We will gain experience with latency-insensitive stream interfaces that implement valid/ready microprotocol, unit testing with test sources/sinks, and using a control/datapath split to implement RTL models. We will also create and use a bit struct to pack request operands into a single message. The code for this section is provided for you in the `tut4_pymtl/gcd` subdirectory.

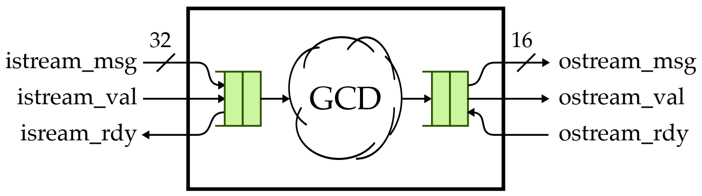


Figure 41: Cloud Diagram for GCD Unit FL Model – Input and output use latency-insensitive stream interfaces; input/output stream queue adapters simplify interacting with these interfaces. The input message includes two 16-bit operands; output message is an 16-bit result.

The previous examples placed the unit test scripts in the same subdirectory as the models these tests were testing. As we start to explore much larger and more complicated designs, it can be useful to keep all of the unit tests together in a separate `test` subdirectory. You can see in this example, that all of the unit tests for the GCD unit are placed in the `tut4_pymtl/gcd/test` subdirectory.

6.1. FL Model of GCD Unit

As before, we begin by designing an FL model of our target GCD unit. Figure 41 shows a cloud diagram for the GCD unit FL model. The GCD unit will take two 16-bit operands and produce a 16-bit result. A key feature of the GCD unit is its use of latency-insensitive stream interfaces to manage flow control for the requests and responses. The interface for the registered incrementer in Section 4 included no extra control signals. A module that wants to use the registered incrementer must explicitly handle the fact that the unit always takes exactly one cycle. The interface for the sorter in Section 5 included an extra valid signal. A module that wants to use the sorter could be carefully constructed so as to be agnostic to the latency of the sorter; this would enable flexibly trying out different sorting algorithms. One issue with including just a valid signal is that there is no way to know if the sorter is busy, and there is no way to tell the sorter that we are not ready to accept the result. In other words, there is no provision for *back pressure*. As shown in Figure 41, our GCD design will use a fully latency-insensitive interface by including two extra signals: a valid and a ready signal. These signals will allow additional flexibility: the GCD unit can indicate it is not ready to accept a new GCD input, and another module can indicate that it is not ready to accept the GCD output.

Assume we have a producer that wishes to send a message to a consumer using the val/rdy microprotocol. At the beginning of the cycle, the producer determines if it has a new message to send to the consumer. If so, it sets the message bits appropriately and then sets the valid signal high. Also at the beginning of the cycle, the consumer determines if it is able to accept a new message from the producer. If so, it sets the ready signal high. At the end of the cycle, the producer and consumer can independently AND the valid and ready signals together; if both signals are true then the message is considered to have been sent from the producer to the consumer and both sides can update their internal state appropriately. Otherwise, we will try again on the next cycle. To avoid long combinational paths and/or combinational loops, we should avoid making the valid signal depend on the ready signal or the ready signal depend on the valid signal. If you absolutely must, you can make the ready signal depend on the valid signal (e.g., in an arbiter) but it is considered very bad practice to make the valid signal depend on the ready signal. As long as you adhere to this policy, composing modules via the stream val/rdy interface should not cause significant timing issues.

Based on the discussion so far, the benefit of a latency-insensitive stream val/rdy interface should be obvious. This interface will allow true black-box testing and will allow flexibly composing modules without regards for the detailed timing properties of each module. For example, if we use the GCD unit in a larger design we can later decide to try a different GCD implementation (with potentially a very different latency), and the larger design should need no modifications! We will use this kind of interface extensively throughout the course.

```
 1 #=========================================================================
 2 # GCD Unit FL Model
 3 #=========================================================================
 4
 5 from math import gcd
 6
 7 from pymtl3 import *
 8 from pymtl3.stdlib.stream.ifcs import IStreamIfc, OStreamIfc
 9 from pymtl3.stdlib.stream      import IStreamDeqAdapterFL, OStreamEnqAdapterFL
10
11 #-------------------------------------------------------------------------
12 # GcdUnitFL
13 #-------------------------------------------------------------------------
14
15 class GcdUnitFL( Component ):
16
17   # Constructor
18
19   def construct( s ):
20
21     # Interface
22
23     s.istream = IStreamIfc( Bits32 )
24     s.ostream = OStreamIfc( Bits16 )
25
26     # Queue Adapters
27
28     s.istream_q = IStreamDeqAdapterFL( Bits32 )
29     s.ostream_q = OStreamEnqAdapterFL( Bits16 )
30
31     s.istream //= s.istream_q.istream
32     s.ostream //= s.ostream_q.ostream
33
34     # FL block
35
36     @update_once
37     def block():
38       if s.istream_q.deq.rdy() and s.ostream_q.enq.rdy():
39         msg = s.istream_q.deq()
40         s.ostream_q.enq( gcd( msg[16:32], msg[0:16] ) )
41
42   # Line tracing
43
44   def line_trace( s ):
45     return f"{s.istream}(){s.ostream}"
```

Figure 42: Gcd Unit FL Model – FL model of greatest-common divisor unit.

In Figure 41, we can see that we often use queues adapters to simplify designing FL models that interact with stream val/rdy interfaces. Figure 42 shows how to implement an FL model for the GCD unit in PyMTL3. The actual work of the FL model takes place on lined 38–40. We use the `gcd` function from the standard Python `math` module to calculate the GCD of the two input operands. This example illustrates two important new features, interfaces and interface adapters.

Lines 23–24 of Figure 42 use interfaces instead of ports as the interface for our GCD unit. We are using gcd message classes to instantiate the interfaces. An RTL interface is simply a collection of logically

```
 1  #=========================================================================
 2  # GcdUnitMsg
 3  #=========================================================================
 4
 5  from pymtl3 import *
 6
 7  #-------------------------------------------------------------------------
 8  # GcdUnitReqMsg
 9  #-------------------------------------------------------------------------
10  # BitStruct designed to hold two operands
11
12  @bitstruct
13  class GcdUnitReqMsg:
14    a: Bits16
15    b: Bits16
16
17  # Usage: GcdUnitMsgs.req, GcdUnitMsgs.resp
18
19  class GcdUnitMsgs:
20    req  = GcdUnitReqMsg
21    resp = Bits16
```

Figure 43: Gcd Unit Message Types – A custom bit struct type is created to hold two operands, and the GcdUnitMsgs collects both request and response type.

related ports (potentially in different directions), which can then be connected in a single statement. For our GCD unit, we are using the `IStreamIfc` and `OStreamIfc` from `stdlib.stream.ifcs` which implement the aforementioned val/rdy microprotocol. This RTL interface groups together the valid, ready, and message ports. Figure 44 shows how the input stream interface is defined in `stdlib`. An RTL interface is just a Python class that inherits from the `Interface` base class provided by the PyMTL3 framework. In the constructor, we define the ports that make up the interface (lines 18–20). We also define methods for converting the interface to a string for simplified line tracing. Figure 43 shows the source code of `GcdUnitMsgs.py`. Line 12–15 defines the bit struct, and Lines 19–21 defines a `GcdUnitMsgs` class that holds the request type and response type.

Lines 28–29 of Figure 42 instantiate two stream queue adapters provided by the PyMTL3 standard library. Interface adapters take the data type as constructor arguments, and then enable the logic within the component to interact with these ports through methods. A `IStreamDeqAdpaterFL` connects to an `IStreamIfc` and provides a standard Python `deq` method for the FL model to use. A `OStreamEnqAdapterFL` connects to an `OStreamIfc` and provides a standard Python `enq` method for the FL model to use. The FL model is responsible for calling the appropriate `rdy` method before calling `deq` or `enq`. We also make use of the `update_once` blocks that are meant to be only called once in each clock cycle. `update_once` blocks are supposed to call methods and modify signals using `@=` blocking assignments. The `update_once` block first invokes `deq.rdy()` method to check if the input stream queue adapter has a message to pop and `enq.rdy()` to check if the output stream queue adapter has available slots to accept a message. If both are ready, we can dequeue a request message from the input stream queue adapter (line 39) and enqueue the response message to the on the output interface (line 40). These queue adapters significantly simplify implementing FL models, since we only need to implement the functionality using method calls.

The PyMTL3 model is in `GcdUnitFL.py` and the corresponding test script is in `GcdUnitFL_test.py` in the `test` subdirectory. One of the nice features of using a latency-insensitive stream interface is that it enables us to use a common framework for sending messages into the device-under-test

```python
#=========================================================================
# ifcs.py
#=========================================================================
# RTL val/rdy interface

from pymtl3 import *

def valrdy_to_str( msg, val, rdy, trace_len=15 ):
  if     val and not rdy: return "#".ljust( trace_len )
  if not val and     rdy: return " ".ljust( trace_len )
  if not val and not rdy: return ".".ljust( trace_len )
  return f"{msg}".ljust( trace_len ) # val and rdy

class IStreamIfc( Interface ):

  def construct( s, Type ):

    s.msg = InPort( Type )
    s.val = InPort()
    s.rdy = OutPort()

    if isinstance( Type, int ):
      s.trace_len = len(str(mk_bits(Type)()))
    else:
      s.trace_len = len(str(Type()))

  def __str__( s ):
    return valrdy_to_str( s.msg, s.val, s.rdy, s.trace_len )

  def line_trace( s ):
    return str( s )
```

Figure 44: Input Stream Interface from `stdlib.stream.ifcs` – Parameterized interfaces that group together the valid, ready, and message ports.

(DUT) and then verifying that the correct messages come out of the DUT. `stdlib.stream` includes the `StreamSourceFL` and `StreamSinkFL` models for this purpose. Figure 45 illustrates the test harness included in the GCD unit test script. We instantiate a test source and attach it to the GCD unit's input stream interface, and then we instantiate a test sink and attach it to the GCD unit's output stream interface. Figure 46 illustrates the overall connectivity in the test harness. Notice how interfaces enable us to connect three ports with a single connect statement (lines 17–18). The test source includes the ability to delay the messages going into the DUT and the test sink includes the ability to apply back-pressure to the DUT. More specifically we can set an *initial delay* (i.e., how many cycles to delay a message after reset) and an *interval delay* (i.e., how many cycles after receiving a message to delay the next message). By using various combinations of these delays we can more robustly ensure that our flow-control logic is working correctly. Note that these test cases illustrate both *directed black-box* and *randomized black-box* testing strategies. The test cases are black-box since they do not depend on the timing within the DUT.

A common testing strategy is for the very first test-case to use directed source/sink messages with no delays. For example, the first test case for our GCD unit FL model creates a couple of source messages along with the correct sink messages. We can run just this test case like this:

```
% cd ${TUTROOT}/build
```

```python
#-------------------------------------------------------------------------
# TestHarness
#-------------------------------------------------------------------------

class TestHarness( Component ):

  def construct( s, gcd ):

    # Instantiate models

    s.src  = StreamSourceFL( Bits32 )
    s.sink = StreamSinkFL( Bits16 )
    s.gcd  = gcd

    # Connect

    s.src.ostream //= s.gcd.istream
    s.gcd.ostream //= s.sink.istream

  def done( s ):
    return s.src.done() and s.sink.done()

  def line_trace( s ):
    return s.src.line_trace() + " > " + \
           s.gcd.line_trace() + " > " + \
           s.sink.line_trace()
```

Figure 45: Excerpt from Unit Test Script for GCD Unit FL Model – Latency insensitive interfaces enable us to use generic sources and sinks for testing.

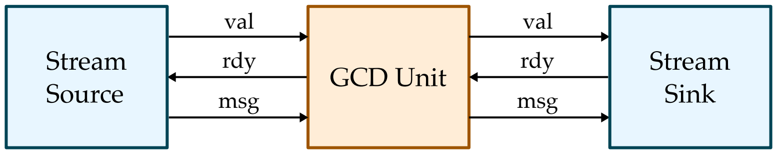


Figure 46: Verifying GCD Using Stream Sources and Sinks – Parameterized stream sources send messages over a stream interface, and parameterized stream sinks receive messages over a stream interface and compare each message to a previously specified reference message.

```
% pytest ../tut4_pymtl/gcd/test/GcdUnitFL_test.py -k basic_0x0 -s
```

Once we know that our design works without any delays, we continue to use directed source/sink messages but then add source delays and sink delays. For example, the second test case for our GCD unit FL model sets the test source to delay the input messages by five cycles. We can also try using no delays on the source, but adding delays to the sink, and finally add delays to both the source and the sink. If we see that our design passes the tests with no delays but fails with delays this is a good indicator that there is an issue with our val/rdy logic.

```
% cd ${TUTROOT}/build
% pytest ../tut4_pymtl/gcd/test/GcdUnitFL_test.py -k basic -s
```

After additional directed testing with delays, we can start to use randomly generated source/sink messages for even greater test coverage.

```
cycle src         A    B     out    sink
---------------------------------------
   7: 09cb:da5d > 09cb:da5d()#    > #
   8:           >          ()#    > #
   9:           >          ()#    > #
  10:           >          ()#    > #
  11: f073:da28 > f073:da28()#    > #
  12: .         > .        ()#    > #
  13: .         > .        ()0001 > 0001
  14: .         > .        ()#    > #
  15: c159:ee21 > c159:ee21()#    > #
  16: .         > .        ()#    > #
  17: .         > .        ()#    > #
  18: .         > .        ()#    > #
  19: #         > #        ()#    > #
  20: #         > #        ()#    > #
  21: #         > #        ()#    > #
  22: #         > #        ()#    > #
```

Figure 47: Line Trace for GCD Unit FL Model – Various characters indicate the status of the val/rdy interface: `.` = val/rdy interface is not valid and not ready; `#` = val/rdy interface is valid but not ready; space = val/rdy interface is not valid and ready; message is shown when it is actually transferred across interface.

```
% cd ${TUTROOT}/build
% pytest ../tut4_pymtl/gcd/test/GcdUnitFL_test.py -k random -s
```

Figure 47 illustrates a portion of the line trace for the randomized testing. Notice that the line trace tells something about what is going on with each val/rdy interface. A period (`.`) indicates that the interface is not ready but also not valid; a hash (`#`) indicates that the interface is valid but not ready; a space indicates that the interface is ready but not valid. The actual message is displayed when it is transferred from the producer to the consumer. We can see a message being sent from the test source into the GCD unit on cycle 7 and although the result is valid on cycle 8 the test sink is not ready until cycle 13 to accept the result. On cycles 8–10 the test source does not have a new message to send to the GCD unit. On cycle 12 it does indeed have a new message, but the GCD unit is not ready because it is still waiting on the test sink. Finally, on cycle 13 the test sink is ready and the GCD unit is able to send the result and accept a new input.

★ *To-Do On Your Own:* Write a new test case for the GCD unit FL model. First create a new list of messages named `coprime_msgs` which includes a few sets of relatively prime numbers. Two numbers are relatively prime (or coprime) if their greatest common divisor is one. Then add two new test cases to the test case table. Both test cases should use `coprime_msgs`. The first new test case should have no delays, and the second new test case should have delays.

6.2. CL Model of GCD Unit

Once we have a reasonable FL model, we can manually refine this model into a CL model. This process often requires exploring different algorithms that can achieve the functional-level behavior yet still be efficiently implemented in hardware. We can implement these algorithms in the CL model, along with a cycle-approximate timing model, to explore the system-level performance impact of different algorithms. Figure 31 illustrates the CL model using a cloud diagram. The high-level approach is to use the first cycle to calculate the GCD and also to estimate the number of cycles a specific algorithm will take. We can then to delay the result some number of cycles to model the cycle-level performance of the target hardware. Unlike the pipelined CL timing model in Section 5, our GCD unit will be using an iterative CL timing model. This means that we do not need to model pipelining multiple results, but instead we just need to wait a certain number of cycles.

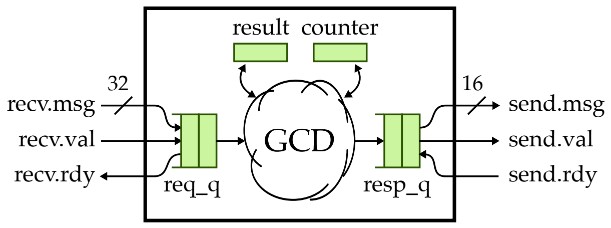


Figure 48: Cloud Diagram for GCD Unit CL Model – CL model uses input/output queue adapters and extra state to create a cycle-level timing model.

Figure 49 shows an excerpt from the CL model for the GCD unit. Lines 3–12 define a helper function that implements the specific algorithm we will be using to calculate the GCD and also estimates the number of cycles this algorithm will take. For now, we have chosen to explore Euclid's algorithm, and we are assuming each iteration of the while loop will take one cycle. This is a reasonable cycle-approximate model for a simple FSM-based RTL model. It would be relatively straight-forward to include multiple algorithms (each with their own timing model) and then to choose a specific algorithm based on a parameter. As in the GCD unit FL model, we are using stream interfaces and queue adapters.

The queue adapters are almost identical to the FL model, except that we don't check the rdy methods at the same time. These queue adapters might introduce extra buffering that may (or may not) be present in the target hardware. This will impact the cycle-level performance. This is a common trade-off we often make when designing CL models; we sometimes reduce the cycle-level accuracy of our CL model in order to simplify the design and enable easier design-space exploration.

The implementation is also relatively straightforward. `s.result` holds the delayed response which is calculated immediately after dequeuing a request. If the unit is "working" on a request, line 45–56 will executed to handle the delay or send out the response when the delay goes down to zero.

Figure 50 shows the unit test script for our GCD unit CL model. Lines 19–29 use directed testing for just the algorithm and the associated timing model. Line 13 imports the test harness and test case table from the GCD unit FL model's test script. We then simply apply the same FL test cases to our GCD unit CL model on lines 35–50. If we add new test cases for the FL model, then they will also be automatically applied to the CL model. Notice how compact the test script is compared to `GcdUnitFL_test.py`. Latency-insensitive stream interfaces combined with the flexibility of the `pytest` framework enable reusing tests across different models. This is an incredibly useful feature and significantly simplifies test-driven development. Lines 40–48 sets the parameter of source and sink using the `set_param` API which enables us to pass in parameters without passing down arguments throught every constructor.

The PyMTL3 model is in `GcdUnitCL.py` and the corresponding test script is in `GcdUnitCL_test.py`. We can run all of the tests and display the line trace for the basic test case with delays in the test sink like this:

```
% cd ${TUTROOT}/build
% pytest ../tut4_pymtl/gcd/test/GcdUnitCL_test.py -v
% pytest ../tut4_pymtl/gcd/test/GcdUnitCL_test.py -sv -k basic_0x5
```

Figure 51 shows the beginning of the line trace for the basic test case. The first GCD request enters the GCD unit on cycle 4 and the response is returned on cycle 10, for a total latency of eight cycles. However, notice that the second GCD request is able to enter the GCD unit right away on cycle 5 even though the first GCD transaction is not done. This is a result of the extra buffering in the queue interface adapters. The second GCD response is sent to the test sink on cycle 18. The third GCD

```python
# gcd algorithm and timing model

def gcd( a, b ):
  ncycles = 0
  while True:
    ncycles += 1
    if a < b:
      a,b = b,a
    elif b != 0:
      a = a - b
    else:
      return (a,ncycles)

class GcdUnitCL( Component ):

  # Constructor

  def construct( s ):

    # Interface

    s.istream = IStreamIfc( GcdUnitMsgs.req  )
    s.ostream = OStreamIfc( GcdUnitMsgs.resp )

    # Queue Adapters

    s.istream_q = IStreamDeqAdapterFL( GcdUnitMsgs.req  )
    s.ostream_q = OStreamEnqAdapterFL( GcdUnitMsgs.resp )

    s.istream //= s.istream_q.istream
    s.ostream //= s.ostream_q.ostream

    # Member variables

    s.result  = None
    s.counter = 0

    # CL block

    @update_once
    def block():

      if s.result is not None:
        # Handle delay to model the gcd unit latency
        if s.counter > 0:
          s.counter -= 1
        elif s.ostream_q.enq.rdy():
          s.ostream_q.enq( s.result )
          s.result = None

      elif s.istream_q.deq.rdy():
        msg = s.istream_q.deq()
        s.result, s.counter = gcd_cl(msg.a, msg.b)

  # Line tracing

  def line_trace( s ):
    return f"{s.recv}({s.counter:^4}){s.send}"
```

Figure 49: Excerpt from Gcd Unit CL Model – CL model of greatest-common divisor unit corresponding to Figure 48.

```
#=========================================================================
# GcdUnitCL_test
#=========================================================================

import pytest

from pymtl3 import *
from pymtl3.stdlib.test_utils import run_sim
from tut4_pymtl.gcd.GcdUnitCL import gcd_cl, GcdUnitCL

# Reuse cases from FL tests

from tut4_pymtl.gcd.test.GcdUnitFL_test import TestHarness, test_case_table

#-------------------------------------------------------------------------
# test_gcd_cl
#-------------------------------------------------------------------------

def test_gcd_cl_calc():
  #              a   b         result ncycles
  assert gcd_cl( 0,  0  ) == ( 0,     1       )
  assert gcd_cl( 1,  0  ) == ( 1,     1       )
  assert gcd_cl( 0,  1  ) == ( 1,     2       )
  assert gcd_cl( 5,  5  ) == ( 5,     3       )
  assert gcd_cl( 15, 5  ) == ( 5,     5       )
  assert gcd_cl( 5,  15 ) == ( 5,     6       )
  assert gcd_cl( 7,  13 ) == ( 1,     13      )
  assert gcd_cl( 75, 45 ) == ( 15,    8       )
  assert gcd_cl( 36, 96 ) == ( 12,    10      )

#-------------------------------------------------------------------------
# Test cases
#-------------------------------------------------------------------------

@pytest.mark.parametrize( **test_case_table )
def test_gcd_cl( test_params ):

  th = TestHarness( GcdUnitCL() )

  th.set_param("top.src.construct",
    msgs=test_params.msgs[::2],
    initial_delay=test_params.src_delay,
    interval_delay=test_params.src_delay )

  th.set_param("top.sink.construct",
    msgs=test_params.msgs[1::2],
    initial_delay=test_params.sink_delay,
    interval_delay=test_params.sink_delay )

  run_sim( th )
```

Figure 50: Unit Test Script for GCD Unit CL Model – We use directed testing for the GCD algorithm and timing model, and reuse the test cases from the GCD unit CL model.

```
cycle src        A    B     out    sink
--------------------------------------
  3:           >          ( 0  ).    > .
  4: 000f:0005 > 000f:0005( 5  ).    > .
  5: 0003:0009 > 0003:0009( 4  ).    > .
  6: #         > #        ( 3  ).    > .
  7: #         > #        ( 2  ).    > .
  8: #         > #        ( 1  ).    > .
  9: #         > #        ( 0  )     >
 10: #         > #        ( 0  )0005 > 0005
 11: #         > #        ( 6  ).    > .
 12: 0000:0000 > 0000:0000( 5  ).    > .
 13: #         > #        ( 4  ).    > .
 14: #         > #        ( 3  ).    > .
 15: #         > #        ( 2  ).    > .
 16: #         > #        ( 1  )     >
 17: #         > #        ( 0  )     >
 18: #         > #        ( 0  )0003 > 0003
 19: #         > #        ( 1  ).    > .
 20: 001b:000f > 001b:000f( 0  ).    > .
 21: #         > #        ( 0  )#    > #
 22: #         > #        ( 10 )#    > #
 23: 0015:0031 > 0015:0031( 9  )#    > #
 24: #         > #        ( 8  )0000 > 0000
 25: #         > #        ( 7  ).    > .
```

Figure 51: Line Trace for CL Implementation of GCD – Extra buffering means the GCD unit can accept the second transaction before the first transaction is done. Recall that various characters indicate the status of the val/rdy interface: `.` = val/rdy interface is not valid and not ready; `#` = val/rdy interface is valid but not ready; space = val/rdy interface is not valid and ready; message is shown when it is actually transferred across interface.

request stalls until cycle 12 when it can enter the GCD unit. On cycle 21, the third GCD response is valid but it cannot be sent to the test sink, since the test sink is not ready (due to a sink delay). The GCD unit must wait until the test sink is ready on cycle 24.

★ *To-Do On Your Own:* It should be possible to do a swap and the following subtract in a single cycle. Modify the timing model to account for this optimization and rerun the test cases to observe how this change impacts the cycle-level performance.

6.3. RTL Model of GCD Unit

When implementing more complicated RTL modaels, we will usually divide the design into two parts: the *datapath* and the *control unit*. The datapath contains the arithmetic operators, muxes, and registers that work on the data, while the control unit is responsible for controlling these components to achieve the desired functionality. The control unit sends *control signals* to the datapath and the datapath sends *status signals* back to the control unit. Figure 52 illustrates the datapath for the GCD unit and Figure 53 illustrates the corresponding finite-state-machine (FSM) control unit. The PyMTL3 source for the datapath, control unit, and the top-level module which composes the datapath and control unit is in `GcdUnitRTL.py`.

Figure 54 shows the interface for the datapath and the first two datapath components. Notice how we use a very structural implementation that *exactly* matches the datapath diagram in Figure 52. We leverage several modules from `stdlib.primitive` (e.g., `Mux`, `RegEn`). You should use a similar structural approach when building your own datapaths for this course. Line 40 shows how we can create a short-hand name for a model (`m`) which further simplifies the syntax for connections. For a net that moves data from right to left in the datapath diagram, we need to declare a dedicated wire right before it is used as an input (e.g., `s.sub_out` and `s.b_reg_out`).

Take a look at the control unit in `GcdUnitRTL.py` and notice the stylized way we write FSMs. An FSM-based control unit should have three parts: a register for the state, an `update_ff` concurrent

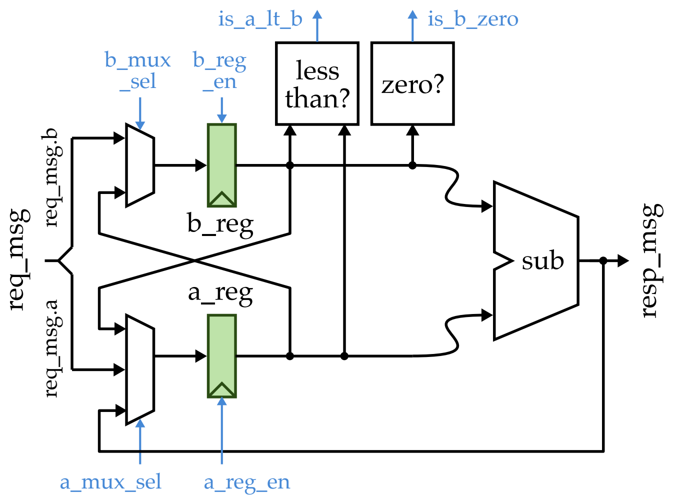


Figure 52: Datapath Diagram for GCD – Datapath includes two state registers and required muxing and arithmetic units to iteratively implement Euclid's algorithm.

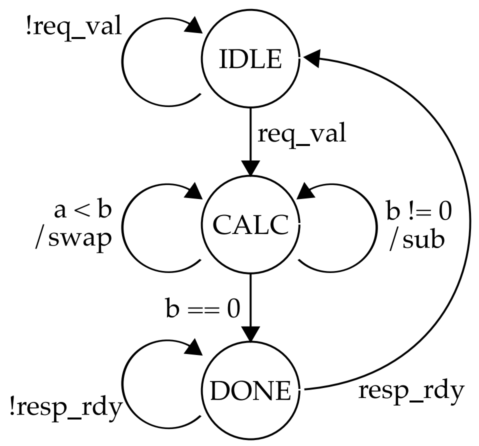


Figure 53: FSM Diagram for GCD – A hybrid Moore/Mealy FSM for controlling the datapath in Figure 52. Mealy transitions in the calc state determine whether to swap or subtract.

block for the state transitions, and an `update` concurrent block for the state outputs. We use `if` statements in both concurrent block to determine the next state and the state outputs based on the current state.

Also take a look at the top-level module which composes the datapath and control unit.

The PyMTL3 model is in `GcdUnitRTL.py` and the corresponding test script is in `GcdUnitRTL_test.py`. As with the GCD unit CL model, our RTL model is able use the exact same test setup as the GCD unit FL model, even though the FL, CL, and RTL models all take different amounts of time to calculate the GCD. This illustrates the power of using latency-insensitive interfaces. We can run all of the tests and display the line trace for the basic test case with delays in the test sink like this:

```
% cd ${TUTROOT}/build
% pytest ../tut4_pymtl/gcd/test/GcdUnitRTL_test.py -v
% pytest ../tut4_pymtl/gcd/test/GcdUnitRTL_test.py -sv -k basic_0x0
```

Figure 55 shows the beginning of the line trace for the basic test case. We use the line trace to show the state of the A and B registers at the beginning of each cycle and use specific characters to indicate which state we are in (i.e., `I` = idle, `Cs` = calc with swap, `C-` = calc with subtract, `D` = done). We can see that the test source sends a new message into the GCD unit on cycle 4. The GCD unit is in the idle state and transitions into the calc state. It does two swaps, three subtractions, and one final calc state before transitioning into the done state on cycle 10. This very first GCD request takes eight cycles. Notice that the second GCD request stalls until the first request is done. The second GCD response is sent to the test sink on cycle 18. Compare this to the line trace from our GCD unit CL model shown in Figure 51. Notice that the extra buffering in the CL model means that the second GCD response is sent to the test sink one cycle too early, and thus the second GCD response is returned on cycle 17 instead of cycle 18. The extra buffering in the output queue adapter can also result in timing discrepancies between the CL and RTL models. We can see now that our GCD unit CL model is a cycle-approximate CL model; while it reasonably reflects the cycle-level behavior of the RTL model it is not cycle accurate.

```
#=========================================================================
# GCD Unit RTL Datapath
#=========================================================================

class GcdUnitDpathRTL(Component):

  # Constructor

  def construct( s ):

    #---------------------------------------------------------------------
    # Interface
    #---------------------------------------------------------------------

    s.req_msg_a = InPort (16)
    s.req_msg_b = InPort (16)
    s.resp_msg  = OutPort(16)

    # Control signals (ctrl -> dpath)

    s.a_mux_sel = InPort( A_MUX_SEL_NBITS )
    s.a_reg_en  = InPort()
    s.b_mux_sel = InPort( B_MUX_SEL_NBITS )
    s.b_reg_en  = InPort()

    # Status signals (dpath -> ctrl)

    s.is_b_zero = OutPort()
    s.is_a_lt_b = OutPort()

    #---------------------------------------------------------------------
    # Structural composition
    #---------------------------------------------------------------------

    # A mux

    s.sub_out   = Wire(16)
    s.b_reg_out = Wire(16)

    s.a_mux = m = Mux( Bits16, 3 )
    m.sel                //= s.a_mux_sel
    m.in_[A_MUX_SEL_IN ] //= s.req_msg_a
    m.in_[A_MUX_SEL_SUB] //= s.sub_out
    m.in_[A_MUX_SEL_B  ] //= s.b_reg_out

    # A register

    s.a_reg = m = RegEn(Bits16)
    m.en  //= s.a_reg_en
    m.in_ //= s.a_mux.out
```

Figure 54: Excerpt from Datapath in GCD Unit RTL Model – We use top-level constants for various control signal encodings (e.g., `A_MUX_SEL_NBITS`, `A_MUX_SEL_IN`), and we use **@=** operator to enable more succinct structural composition in datapaths.

```
cycle src        A    B    Areg            Breg           ST out    sink
---------------------------------------------------------------------------
   3:           >         ([en|0000 > 0000] [en|0000 > 0000] I )     >
   4: 000f:0005 > 000f:0005([en|000f > 0000] [en|0005 > 0000] I )     >
   5: #         > #       ([en|000a > 000f] [  |000f > 0005] C-)     >
   6: #         > #       ([en|0005 > 000a] [  |000a > 0005] C-)     >
   7: #         > #       ([en|0000 > 0005] [  |0005 > 0005] C-)     >
   8: #         > #       ([en|0005 > 0000] [en|0000 > 0005] Cs)     >
   9: #         > #       ([en|0005 > 0005] [  |0005 > 0000] C )     >
  10: #         > #       ([  |0003 > 0005] [  |0005 > 0000] D )0005 > 0005
  11: 0003:0009 > 0003:0009([en|0003 > 0005] [en|0009 > 0000] I )     >
  12: #         > #       ([en|0009 > 0003] [en|0003 > 0009] Cs)     >
  13: #         > #       ([en|0006 > 0009] [  |0009 > 0003] C-)     >
  14: #         > #       ([en|0003 > 0006] [  |0006 > 0003] C-)     >
  15: #         > #       ([en|0000 > 0003] [  |0003 > 0003] C-)     >
  16: #         > #       ([en|0003 > 0000] [en|0000 > 0003] Cs)     >
  17: #         > #       ([en|0003 > 0003] [  |0003 > 0000] C )     >
  18: #         > #       ([  |0000 > 0003] [  |0003 > 0000] D )0003 > 0003
  19: 0000:0000 > 0000:0000([en|0000 > 0003] [en|0000 > 0000] I )     >
  20: #         > #       ([en|0000 > 0000] [  |0000 > 0000] C )     >
  21: #         > #       ([  |001b > 0000] [  |0000 > 0000] D )0000 > 0000
  22: 001b:000f > 001b:000f([en|001b > 0000] [en|000f > 0000] I )     >
```

Figure 55: Line Trace for RTL Implementation of GCD – State of A and B registers at the beginning of the cycle is shown, along with the current state of the FSM. `I` = idle, `Cs` = calc with swap, `C-` = calc with subtract, `D` = done. Recall that various characters indicate the status of the val/rdy interface: `.` = val/rdy interface is not valid and not ready; `#` = val/rdy interface is valid but not ready; space = val/rdy interface is not valid and ready; message is shown when it is actually transferred across interface.

★ *To-Do On Your Own:* Optimize the GCD implementation to improve the performance on the given input datasets.

A first optimization would be to transition into the done state if either a *or* b are zero. If a is zero and b is greater than zero, we will swap a and b and then end the calculation on the next cycle anyways. You will need to carefully modify the datapath and control so that the response can come from either the a or b register.

A second optimization would be to avoid the bubbles caused by the IDLE and DONE states. First, add an edge from the CALC state directly back to the IDLE state when the calculation is complete and the response interface is ready. You will need to carefully manage the response valid bit. Second, add an edge from the CALC state back to the CALC state when the calculation is complete, the response interface is ready, and the request interface is valid. These optimizations should eliminate any bubbles and improve the performance of back-to-back GCD transactions.

A third optimization would be to perform a swap and subtraction in the same cycle. This will require modifying both the datapath and the control unit, but should have a significant impact on the overall performance. Consider the effort required to explore this optimization in the CL model vs. the RTL model.

6.4. Exploring the GCD Implementation

As in the previous section, you can test the translated Verilog using the `--test-verilog` command line option to `pytest`:

```
% cd ${TUTROOT}/build
% pytest ../tut4_pymtl/gcd --test-verilog
```

We have also provided you with a simulator script to evaluate the performance of the GCD implementations. You can run the simulators and look at the average number of cycles to compute a GCD for each input dataset like this:

```
% cd ${TUTROOT}/build
% ../tut4_pymtl/gcd/gcd-sim --stats --impl cl  --input random
% ../tut4_pymtl/gcd/gcd-sim --stats --impl rtl --input random
```

Notice that since our GCD unit CL model is a cycle-approximate model, the total number of cycles for the two models does not need to match exactly, although in this case they do match. You can generate the Verilog and waveforms to drive an FPGA or ASIC toolflow using the simulator like this:

```
% cd ${TUTROOT}/build
% ../tut4_pymtl/gcd/gcd-sim --impl rtl --input random --translate --dump-vcd
% ../tut4_pymtl/gcd/gcd-sim --impl rtl --input small  --translate --dump-vcd
% ../tut4_pymtl/gcd/gcd-sim --impl rtl --input zeros  --translate --dump-vcd
```

Acknowledgments

This tutorial was developed for the ECE 4750 Computer Architecture and ECE 5745 Complex Digital ASIC Design courses at Cornell University by Christopher Batten. The PyMTL hardware modeling framework was developed primarily by Derek Lockhart at Cornell University, and this development was supported in part by NSF CAREER Award #1149464, a DARPA Young Faculty Award, and donations from Intel Corporation and Synopsys, Inc. The PyMTL3 hardware modeling framework was developed by Shunning Jiang, Peitian Pan, and Yanghui Ou. This work was supported in part by NSF CRI Award #1512937, DARPA POSH Award #FA8650-18-2-7852, a research gift from Xilinx, Inc., and the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA, as well as equipment, tool, and/or physical IP donations from Intel, Xilinx, Synopsys, Cadence, and ARM.

Appendix A: Constructs Allowed in Synthesizable Concurrent Blocks

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.