ECE 4750 Computer Architecture, Fall 2016
Tutorial 3: PyMTL Hardware Modeling Framework

School of Electrical and Computer Engineering
Cornell University

revision: 2016-09-06-17-35

1 Introduction 3

2 PyMTL Modeling: Functional-, Cycle-, and Register-Transfer-Level Modeling 4

2.1 Comparison of FL, CL, and RTL Modeling . . . 4

2.2 Synthesizable vs. Non-Synthesizable RTL Modeling . . . 4

3 PyMTL Basics: Data Types and Operators 5

3.1 `Bits` Data Type . . . 5

3.2 `Bits` Operators . . . 7

3.3 `BitStruct` Data Type . . . 11

4 Registered Incrementer 11

4.1 Modeling a Registered Incrementer . . . 11

4.2 Simulating a Model . . . 14

4.3 Visualizing a Model with Line Traces . . . 16

4.4 Visualizing a Model with Waveforms . . . 16

4.5 Verifying a Model with Unit Testing . . . 17

4.6 Verifying a Model with Test Vectors . . . 21

4.7 Verifying a Model with Random Testing . . . 24

4.8 Reusing a Model with Structural Composition . . . 25

4.9 Parameterizing a Model with "Static" Elaboration . . . 28

4.10 Packaging a Collection of Models . . . 32

5 Sort Unit 34

5.1 FL Model of Sort Unit . . . 34

5.2 CL Model of Sort Unit . . . 36

5.3 Flat RTL Model of Sort Unit . . . 39

5.4 Structural RTL Model of Sort Unit . . . 42

5.5 Evaluating Sort Unit using a Simulator . . . 43

5.6 Translating RTL Model of Sort Unit to Verilog . . . 45

6 Greatest Common Divisor Unit **49**

6.1 FL Model of GCD Unit 49

6.2 CL Model of GCD Unit 54

6.3 RTL Model of GCD Unit 57

6.4 Exploring the GCD Implementation 61

7 TravisCI for Continuous Integration **61**

1. Introduction

In the lab assignments for this course, we will be using the PyMTL hardware modeling framework for functional-level modeling, verification, and simulator harnesses. Students can choose to use either PyMTL or Verilog for their register-transfer-level modeling. If you are planning to use Verilog, you should still complete this tutorial since we will always be using PyMTL for some aspects of the lab assignment.

This tutorial describes the basics of the PyMTL hardware modeling framework with a focus on the specific development, testing, and evaluation approach as well as the coding conventions we will be using in the course. We will be using several open-source packages and tools: the `py.test` framework for powerful test-driven Python development; Verilator (`verilator`) for converting Verilog models into C++ source code; and GTKWave (`gtkwave`) for viewing waveforms. The PyMTL framework is itself open source and available on GitHub here:

- `https://github.com/cornell-brg/pymtl`

You should feel free to browse the source code for PyMTL on GitHub if you want to see more how various aspects of the framework are implemented. These tools are installed and available on `ecelinux`. 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 PyMTL by Derek Lockhart et al. titled "PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research" and published at the 47th ACM/IEEE Int'l Symp. on Microarchitecture (MICRO-47). Both of these resources are available on the course website.

To follow along with the tutorial, access the course computing resources, and 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-ece4750.sh script** or that you have added it to your `.bashrc` script, which will then source the script every time you login. Sourcing the setup script sets up the environment required for this class.

You should start by forking the tutorial repository on GitHub. Go to the GitHub page for the tutorial repository located here:

- `https://github.com/cornell-ece4750/ece4750-tut3-pymtl`

Click on *Fork* in the upper right-hand corner. If asked where to fork this repository, choose your personal GitHub account. After a few seconds, you should have a new repository in your account:

- `https://github.com/<githubid>/ece4750-tut3-pymtl`

Where `<githubid>` is your GitHub ID, not your NetID. Now access `ecelinux` and clone your copy of the tutorial repository as follows:

```
% source setup-ece4750.sh
% mkdir -p ${HOME}/ece4750
% cd ${HOME}/ece4750
% git clone https://github.com/<githubid>/ece4750-tut3-pymtl.git tut3
```

```
% cd tut3/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-ece4750.sh` script, and your specific environment may be different from what is assumed in this tutorial.

2. PyMTL Modeling: 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 PyMTL, FL, CL, and RTL models all use port-based interfaces, concurrent blocks, and structural composition. A set of unified interfaces enables PyMTL 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 PyMTL is embedded within Python, which is a fully general-purpose language. Given this, it is very easy to write PyMTL code that does not actually model any kind of realistic hardware. Indeed, we actually need this feature to be able to write clean and productive functional-level models, test harnesses, assertions, and line tracing. **So students must be very diligent in actively deciding whether or not they are writing synthesizable register-transfer-level models or**

non-synthesizable code. Students must always keep in mind what hardware they are modeling and how they are modeling it!

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

Appendix A includes a table that outlines which Python constructs are allowed in synthesizable PyMTL concurrent blocks, which constructs are allowed in synthesizable PyMTL concurrent blocks with limitations, and which constructs are explicitly not allowed in synthesizable PyMTL concurrent blocks.

3. PyMTL Basics: Data Types and Operators

We will begin by writing some very basic code to explore PyMTL 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 PyMTL framework into the global namespace.

```
% cd ${TUTROOT}
% python
>>> from pymtl 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. PyMTL 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 PyMTL 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 PyMTL 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 these commands into the Python interpreter and observe the output.

The `Bits` constructor takes to two arguments specifying the bitwidth and the initial value. Remember that in Python, a *variable* is just a *name* that refers to a *value* or *object*. So on line 4, we create a new variable with the name `a` that refers to a new `Bits` object with 16 bits 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 5, the type of `a` is `Bits`. We might also say that `a` holds an *instance* of type `Bits`. Lines 9–13 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.

Lines 25–28 show how to use standard Python syntax to specify numeric literals in binary or hexadecimal form. Lines 32–37 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 42–43 illustrate two examples where a positive and negative literal are too large to be stored in just eight bits. Lines 47–50 illustrate the optional `Bits` constructor `trunc` argument that will truncate initial values which are too large to store in the given number of bits.

★ *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  # Bits constructor specifies bitwidth
2  # and initial value
3
4  >>> a = Bits( 16, 37 )
5  >>> type(a)
6  <class 'pymtl.datatypes.Bits.Bits'>
7  >>> a
8  Bits( 16, 0x0025 )
9  >>> a = 47
10 >>> type(a)
11 <type 'int'>
12 >>> a
13 47
14
15 # Getting number of bits and value
16
17 >>> a = Bits( 16, 37 )
18 >>> a.nbits
19 16
20 >>> a.uint()
21 37
22
23 # Using binary and hexadecimal literals
24
25 >>> Bits( 8, 0b10101100 )
26 Bits( 8, 0xac )
27 >>> Bits( 32, 0xabcd0123 )
28 Bits( 32, 0xabcd0123 )
29
30 # Negative values stored in two's complement
31
32 >>> Bits( 8, -1 )
33 Bits( 8, 0xff )
34 >>> Bits( 8, -2 )
35 Bits( 8, 0xfe )
36 >>> Bits( 8, -128 )
37 Bits( 8, 0x80 )
38
39 # Initial values that cannot be stored with
40 # given bitwidth throw an exception
41
42 >>> Bits( 8,  300 )
43 >>> Bits( 8, -300 )
44
45 # Truncating initial values
46
47 >>> Bits( 8, 300, trunc=True )
48 Bits( 8, 0x2c )
49 >>> Bits( 8, 0xdeadbeef, trunc=True )
50 Bits( 8, 0xef )
```

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 23 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 28), then later accesses to that object using either name will reflect this change (line 29 and 31). In other words, simply assigning `a` to `b` on line 23, *does not copy the object*. To copy the object, we must create a new `Bits` object as shown on line 37.

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

<table>
<tr><th>Operator</th><th>Description</th></tr>
<tr><td>&</td><td>bitwise AND</td></tr>
<tr><td>|</td><td>bitwise OR</td></tr>
<tr><td>^</td><td>bitwise XOR</td></tr>
<tr><td>^~</td><td>bitwise XNOR</td></tr>
<tr><td>~</td><td>bitwise NOT</td></tr>
</table>

Arithmetic Operators

Operator	Description
+	addition
-	subtraction
*	multiplication

Reduction Operators

Function	Description
reduce_and	reduce via AND
reduce_or	reduce via OR
reduce_xor	reduce via XOR

Shift Operators

Operator	Description
>>	shift right
<<	shift left

Relational Operators

Operator	Description
==	equal
!=	not equal
>	greater than
>=	greater than or equals
<	less than
<=	less than or equals

Other Functions

Function	Description
concat	concatenate
sext	sign-extension
zext	zero-extension

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 18–22 illustrate support for implicit operand conversion. When operators are applied to a mix of `Bits` objects and standard integer values, PyMTL attempts to implicitly convert the standard integer values into `Bits` objects.

★ *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/XNOR operators and the reduction operators.

```
# Logical operators

>>> a = Bits( 4, 0b1010 )
>>> b = Bits( 4, 0b1100 )
>>> a & b
Bits( 4, 0x8 )  # 0b1000
>>> a | b
Bits( 4, 0xe )  # 0b1110
>>> a ^ b
Bits( 4, 0x6 )  # 0b0110
>>> a ^~ b
Bits( 4, 0x9 )  # 0b1001
>>> ~a
Bits( 4, 0x5 )  # 0b0101

# Implicit operand conversion

>>> a = Bits( 4, 0b1010 )
>>> a & 0b1100
Bits( 4, 0x8 )  # 0b1000
>>> 0b1100 & a
Bits( 4, 0x8 )  # 0b1000

# Reduction operators

>>> a = Bits( 8, 0b10101100 )
>>> reduce_and(a)
Bits( 1, 0x0 )
>>> reduce_or(a)
Bits( 1, 0x1 )
>>> reduce_xor(a)
Bits( 1, 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 = Bits( 4, 0b1011 )
4  >>> a << 2
5  Bits( 4, 0xc )  # 0b1100
6  >>> a >> 2
7  Bits( 4, 0x2 )  # 0b0010
8
9  >>> b = Bits( 8, 2 )
10 >>> a << b
11 Bits( 4, 0xc )  # 0b1100
12 >>> a >> b
13 Bits( 4, 0x2 )  # 0b0010
14
15 # Arithmetic operators
16
17 >>> a = Bits( 4, 3 )
18 >>> a + 2
19 Bits( 4, 0x5 )
20 >>> a + 15
21 Bits( 4, 0x2 )
22 >>> a - 2
23 Bits( 4, 0x1 )
24 >>> a - 15
25 Bits( 4, 0x4 )
26
27 >>> b = Bits( 8, 2 )
28 >>> a + b
29 Bits( 8, 0x05 )
30 >>> a - b
31 Bits( 8, 0x01 )
32
33 >>> c = Bits( 8, -2 )
34 >>> a + c
35 Bits( 8, 0x01 )
36 >>> a - c
37 Bits( 8, 0x05 )
38
39 # Relational operators
40
41 >>> a = Bits( 4, 3 )
42 >>> b = Bits( 4, 2 )
43 >>> a == b
44 False
45 >>> a != b
46 True
47 >>> a > b
48 True
49 >>> a >= b
50 True
51 >>> a < b
52 False
53 >>> a <= b
54 False
```

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 3–8 illustrate concatenating two `Bits` objects using the `concat` function. Lines 10–15 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 19–29 illustrate using the `sext` and `zext` functions to sign extend and zero extend a `Bits` object to the given larger bitwidth.

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

```
 1  # Concatenation
 2
 3  >>> a = Bits( 8,  0xab  )
 4  >>> b = Bits( 12, 0xcde )
 5  >>> concat( a, b )
 6  Bits( 20, 0xabcde )
 7  >>> concat( b, a )
 8  Bits( 20, 0xcdeab )
 9
10  >>> a = Bits( 4,  0xd   )
11  >>> b = Bits( 12, 0xead )
12  >>> c = Bits( 12, 0xbee )
13  >>> d = Bits( 4,  0xf   )
14  >>> concat( a, b, c, d )
15  Bits( 32, 0xdeadbeef )
16
17  # Zero/sign extension
18
19  >>> a = Bits( 4, 0xa )
20  >>> sext( a, 8 )
21  Bits( 8, 0xfa )
22  >>> zext( a, 8 )
23  Bits( 8, 0x0a )
24
25  >>> a = Bits( 4, 0x6 )
26  >>> sext( a, 8 )
27  Bits( 8, 0x06 )
28  >>> zext( a, 8 )
29  Bits( 8, 0x06 )
```

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 3–7 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. We can instantiate new Point objects, read the named fields, and write the named fields. Lines 18–21 illustrate that a BitStruct is also a Bits object so all of the standard Bits operators are available for use with BitStruct objects.

Lines 25–39 define a parameterized BitStruct where the bitwidth of the two coordinate fields is given as a constructor argument. Line 30 shows how we can define a new name for a specific instance of this parameterized BitStruct where each field is eight bits.

★ *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
3  >>> class Point( BitStructDefinition ):
4  ...   def __init__( s ):
5  ...     s.x = BitField(4)
6  ...     s.y = BitField(4)
7  ...
8  >>> pt1 = Point()
9  >>> pt1.x = 3
10 >>> pt1.y = 4
11 >>> pt1
12 Bits( 8, 0x34 )
13 >>> pt1.x
14 Bits( 4, 0x3 )
15 >>> pt1.y
16 Bits( 4, 0x4 )
17
18 >> pt1 & Bits( 8, 0xf0 )
19 Bits( 8, 0x30 )
20 >> pt1[0:4]
21 Bits( 4, 0x4 )
22
23 # Parameterized Point BitStruct
24
25 >>> class PointN( BitStructDefinition ):
26 ...   def __init__( s, nbits ):
27 ...     s.x = BitField(nbits)
28 ...     s.y = BitField(nbits)
29 ...
30 >>> Point8 = PointN(8)
31 >>> pt2 = Point8()
32 >>> pt2.x = 3
33 >>> pt2.y = 4
34 >>> pt2
35 Bits( 16, 0x0304 )
36 >>> pt2.x
37 Bits( 8, 0x03 )
38 >>> pt2.y
39 Bits( 8, 0x04 )
```

Figure 6: Creating and Using BitStruct Objects

4. Registered Incrementer

In this section, we will create our very first PyMTL hardware model and then learn how to simulate, visualize, verify, reuse, parameterize, and package this model. 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 PyMTL 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 PyMTL. Every PyMTL file should begin with a header comment as shown on lines 1–6. The header comment identifies

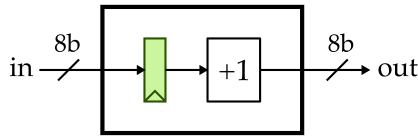

Figure 7: Block Diagram for Registered Incrementer – An eight-bit registered incrementer with an eight-bit input port, an eight-bit output port, and (implicit) clock and reset inputs.

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 PyMTL 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 PyMTL framework on line 8. A PyMTL model is just a Python class that inherits from the `Model` base class provided by the PyMTL framework. A couple of comments about the coding conventions that we will be using in this course. PyMTL 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, since `in` is a Python keyword. 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.

The model's constructor is used to declare the port-based interface, instantiate child models, connect ports, and define concurrent blocks. This simple model does not include any child models 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 models 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 PyMTL framework. The constructor for these port 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 PyMTL models have implicit `clk` and `reset` input ports. PyMTL models should never write the special `clk` or `reset` signal directly, and PyMTL models should never read the `clk` signal. PyMTL models 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 PyMTL "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 `s.tick` 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

```
#=========================================================================
# RegIncr
#=========================================================================
# This is a simple model for a registered incrementer. An eight-bit value
# is read from the input port, registered, incremented by one, and
# finally written to the output port.

from pymtl import *

class RegIncr( Model ):

  # Constructor

  def __init__( s ):

    # Port-based interface

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

    # Concurrent block modeling register

    s.reg_out = Wire( Bits(8) )

    @s.tick
    def block1():
      if s.reset:
        s.reg_out.next = 0
      else:
        s.reg_out.next = s.in_

    # Concurrent block modeling incrementer

    @s.combinational
    def block2():
      s.out.value = s.reg_out + 1
```

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

copy the value on the input port to the `reg_out` wire. When writing signals from within a `s.tick` concurrent block, we always use the `next` attribute. The `next` attribute informs the framework that this write should only be visible after all other `s.tick` concurrent blocks have executed. Using the `next` attribute is the key to making it appear as if all `s.tick` 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 `s.combinational` 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 `s.combinational` 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 a `s.combinational` block never reach a fixed point then we say the design has a "combinational loop." When writing signals from within a `s.combinational` concurrent block, we always use the `value` attribute. Unlike using the `next` attribute, the `value` attribute informs the framework that this write should be visible immediately. The write to the `out` port can cause other `s.combinational` concurrent blocks in other models 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 `s.tick` concurrent blocks in the system. After all `s.tick` 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 `s.tick` concurrent block (i.e., synchronous reset). Reading the reset signal in a `s.combinational` 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 PyMTL source file named `RegIncr.py` `tut3_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. Simulating a Model

Now that we have developed a new hardware model, we can test its functionality using a simulator 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 `SimulationTool` to create a simulator. A key feature of PyMTL is its model/tool split, meaning that designers create models and then use various tools (such as the `SimulationTool`) to manipulate their elaborated designs. We reset the simulator on line 26 which will raise the implicit reset signal for two cycles. Lines 30–43 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 the `value` attribute when writing ports in the simulator script, similar to how signals are written from within `s.combinational` concurrent blocks.

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 simulation tool. Then run the simulator script as follows:

```
% cd ${TUTROOT}/tut3_pymtl/regincr
% ./regincr-sim 0x01 0x13 0x25 0x37
```

You should see output from executing the simulator over several cycles. Note that the output starts on cycle 2; 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.

```
#!/usr/bin/env python
#=========================================================================
# regincr-sim <input-values>
#=========================================================================

from pymtl   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 )

# Elaborate the model

model = RegIncr()
model.elaborate()

# Create and reset simulator

sim = SimulationTool( model )
sim.reset()

# Apply input values and display output values

for input_value in input_values:

  # Write input value to input port

  model.in_.value = input_value

  # Display input and output ports

  print " cycle = {}: in = {}, out = {}" \
    .format( sim.ncycles, model.in_, model.out )

  # Tick simulator one cycle

  sim.cycle()
```

Figure 9: Simulator for Registered Incrementer – Python script to elaborate the model, create a simulator, write input values to the input ports, and display the input/output ports.

★ *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, PyMTL 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` model:

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

Each model'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, replace the `print` statement on lines 38–39 in the `regincr-sim` script shown in Figure 9 with `sim.print_line_trace()`. 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 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 PyMTL framework can output waveforms in the Verilog Change Dump (VCD) format for every signal (i.e., ports and wires) in your design.

To generate VCD, you need to set the `vcd_file` attribute on your model after construction but before elaboration. This attribute should be set to the desired file name for the generated VCD. Add the following after line 20 in the `regincr-sim` script shown in Figure 9:

```
model.vcd_file = "regincr-sim.vcd"
```

Then rerun the simulator script, and use the open-source GTKWave program to browse the generated waveforms as follows:

```
% cd ${TUTROOT}/tut3_pymtl/regincr
% ./regincr-sim 0x01 0x13 0x25 0x37
```

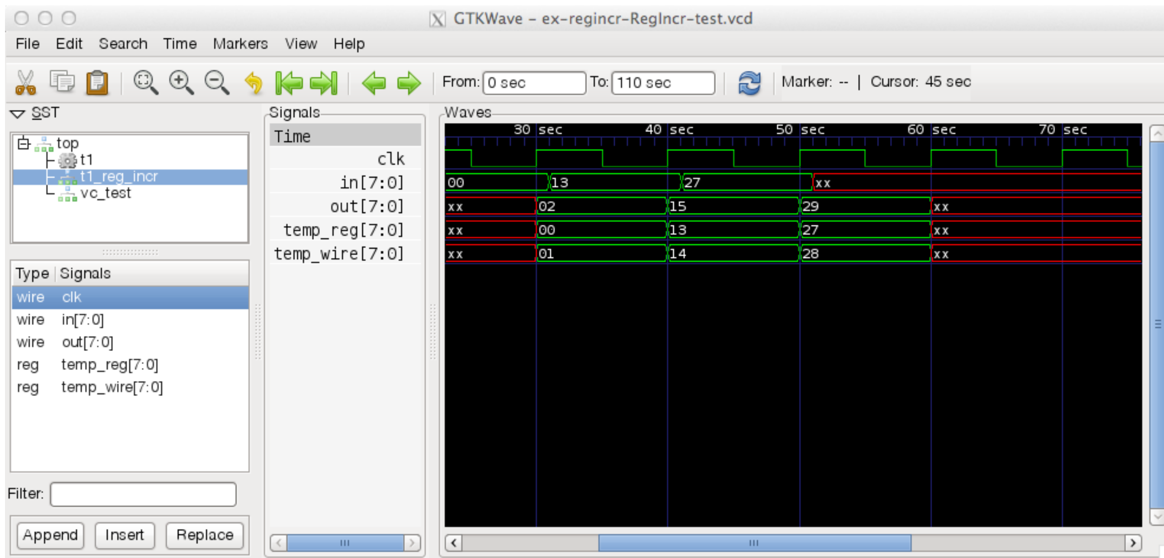

Figure 10: GTKWave Waveform Viewer – GTKWave is being used to browse the signals associated with the registered incrementer shown in Figure 8 and the simulator script shown in Figure 9.

```
% gtkwave regincr-sim.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* 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 10 illustrates using GTKWave to view the waveforms from our simulator script. 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.

4.5. 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 `py.test` unit testing framework. The `py.test` 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 11 illustrates a simple unit testing script for our registered incrementer. Notice at a high-level the test code is very straight-forward; the `py.test` framework enables unit testing to be as simple or as complex as necessary. The `py.test` framework includes automatic test discovery, which means that it will look through the unit test script and assume that any function that begins with `test_` is a test case. In this example, `py.test` will discover a single test case named `test_basic` corresponding to the function declared on lines 13–59. To test our registered incrementer, we need to instantiate and elaborate the model, use the simulation tool to create a simulator, 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 17–19 instantiate and elaborate the model. Note that `dump_vcd` is specified as an argument to the unit test, and then used as the file name for the generated VCD file. PyMTL is setup to treat the `dump_vcd` argument specially. If a user includes `--dump-vcd` on the command-line when running `py.test`, 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 `py.test`, then `dump_vcd` will be `None` and no VCD file will be generated. Lines 23–24 use the `SimulationTool` to create and reset a simulator.

Lines 29–50 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 33 writes the test input to the `in_` port of the registered incrementer. Note that it is important to use the `value` attribute when writing ports in the test harness, similar to how signals are written from within `s.combinational` concurrent blocks. Line 37 tells the simulator to call any `s.combinational` 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 `py.test` framework does not need special assertion checking functions, and instead hooks into the standard `assert` statement provided in Python. This means the `py.test` 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

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

from pymtl   import *
from RegIncr import RegIncr

# In py.test, unit tests are simply functions that begin with a "test_"
# prefix. PyMTL is setup to simplify dumping VCD. Simply specify
# "dump_vcd" as an argument to your unit test, and then you can dump VCD
# with the --dump-vcd option to py.test.

def test_basic( dump_vcd ):

  # Elaborate the model

  model = RegIncr()
  model.vcd_file = dump_vcd
  model.elaborate()

  # Create and reset simulator

  sim = SimulationTool( model )
  sim.reset()
  print ""

  # Helper function

  def t( in_, out ):

    # Write input value to input port

    model.in_.value = in_

    # Ensure that all combinational concurrent blocks are called

    sim.eval_combinational()

    # Display a line trace

    sim.print_line_trace()

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

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

    # Tick simulator one cycle

    sim.cycle()

  # Cycle-by-cycle tests

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

Figure 11: Unit Test Script for Registered Incrementer – A unit test for the eight-bit registered incrementer in Figure 8, which uses the `py.test` unit testing framework.

```
1  ========================= test session starts ==========================
2  platform darwin -- Python 2.7.5 -- py-1.4.26 -- pytest-2.6.4
3  plugins: xdist
4  collected 1 items
5
6  ../tut3_pymtl/regincr/RegIncr_test.py .
7
8  ======================= 1 passed in 0.04 seconds ========================
```

Figure 12: `py.test` **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.

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 11 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 `py.test` framework for automatic test discovery. Add the tests cases shown on lines 54–59 in Figure 8. We can run the test script using `py.test` as follows:

```
% mkdir ${TUTROOT}/build
% cd ${TUTROOT}/build
% py.test ../tut3_pymtl/regincr/RegIncr_test.py
```

Note that we run our unit test scripts from within a separate build directory. The PyMTL framework often creates extra temporary and/or output files, so keeping these generated files in a separate build directory helps avoid creating generated files in the source tree and facilitates performing a clean build. The `py.test` framework automatically discovers the `test_basic` test case. The output from running `py.test` should look similar to what is shown in Figure 12; `py.test` 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 12. This is because by default, the `py.test` 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
% py.test ../tut3_pymtl/regincr/RegIncr_test.py -s
```

Note that by default, `py.test` 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
% py.test ../tut3_pymtl/regincr/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
% py.test ../tut3_pymtl/regincr/RegIncr_test.py --dump-vcd
% gtkwave tut3_pymtl.regincr.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.

4.6. Verifying a Model with Test Vectors

The unit test shown in Figure 11 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 11, then each test case would need to duplicate lines 15–50. 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, PyMTL 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 `pclib` (PyMTL Component 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 `pclib`, you cab browse the source code on the public PyMTL GitHub repository:

- `https://github.com/cornell-brg/pymtl/tree/pclib/rtl`
- `https://github.com/cornell-brg/pymtl/tree/pclib/test`

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

- `https://github.com/cornell-brg/pymtl/blob/pclib/test/test_utils.py#L75-L159`

Test vectors are essentially a table of test inputs and reference outputs. Figure 13 shows an extra test script that uses the `run_test_vector_sim` helper function provided by the PyMTL 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

```
1  #=========================================================================
2  # RegIncr_extra_test
3  #=========================================================================
4
5  from pymtl      import *
6  from pclib.test import run_test_vector_sim
7  from RegIncr    import RegIncr
8
9  #-------------------------------------------------------------------------
10 # test_small
11 #-------------------------------------------------------------------------
12
13 def test_small( dump_vcd ):
14   run_test_vector_sim( RegIncr(), [
15     ('in_   out*'),
16     [ 0x00, '?'  ],
17     [ 0x03, 0x01 ],
18     [ 0x06, 0x04 ],
19     [ 0x00, 0x07 ],
20   ], dump_vcd )
21
22 #-------------------------------------------------------------------------
23 # test_large
24 #-------------------------------------------------------------------------
25
26 def test_large( dump_vcd ):
27   run_test_vector_sim( RegIncr(), [
28     ('in_   out*'),
29     [ 0xa0, '?'  ],
30     [ 0xb3, 0xa1 ],
31     [ 0xc6, 0xb4 ],
32     [ 0x00, 0xc7 ],
33   ], dump_vcd )
34
35 #-------------------------------------------------------------------------
36 # test_overflow
37 #-------------------------------------------------------------------------
38
39 def test_overflow( dump_vcd ):
40   run_test_vector_sim( RegIncr(), [
41     ('in_   out*'),
42     [ 0x00, '?'  ],
43     [ 0xfe, 0x01 ],
44     [ 0xff, 0xff ],
45     [ 0x00, 0x00 ],
46   ], dump_vcd )
```

Figure 13: 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 `py.test` unit testing framework.

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 11. This sophisticated helper function demonstrates the power of using a general-purpose dynamic language such as Python to write test harnesses.

```
1  ========================= test session starts ==========================
2  platform darwin -- Python 2.7.5 -- py-1.4.26 -- pytest-2.6.4
3  plugins: xdist
4  collected 21 items
5
6  ../tut3_pymtl/regincr/RegIncr2stage_test.py::test_small FAILED
7  ../tut3_pymtl/regincr/RegIncr2stage_test.py::test_large FAILED
8  ../tut3_pymtl/regincr/RegIncr2stage_test.py::test_overflow FAILED
9  ../tut3_pymtl/regincr/RegIncr2stage_test.py::test_random FAILED
10 ../tut3_pymtl/regincr/RegIncrNstage_test.py::test[2stage_small] FAILED
11 ../tut3_pymtl/regincr/RegIncrNstage_test.py::test[2stage_large] FAILED
12 ../tut3_pymtl/regincr/RegIncrNstage_test.py::test[2stage_overflow] FAILED
13 ../tut3_pymtl/regincr/RegIncrNstage_test.py::test[2stage_random] FAILED
14 ../tut3_pymtl/regincr/RegIncrNstage_test.py::test[3stage_small] FAILED
15 ../tut3_pymtl/regincr/RegIncrNstage_test.py::test[3stage_large] FAILED
16 ../tut3_pymtl/regincr/RegIncrNstage_test.py::test[3stage_overflow] FAILED
17 ../tut3_pymtl/regincr/RegIncrNstage_test.py::test[3stage_random] FAILED
18 ../tut3_pymtl/regincr/RegIncrNstage_test.py::test_random[1] PASSED
19 ../tut3_pymtl/regincr/RegIncrNstage_test.py::test_random[2] FAILED
20 ../tut3_pymtl/regincr/RegIncrNstage_test.py::test_random[3] FAILED
21 ../tut3_pymtl/regincr/RegIncrNstage_test.py::test_random[4] FAILED
22 ../tut3_pymtl/regincr/RegIncrNstage_test.py::test_random[5] FAILED
23 ../tut3_pymtl/regincr/RegIncrNstage_test.py::test_random[6] FAILED
24 ../tut3_pymtl/regincr/RegIncr_extra_test.py::test_small PASSED
25 ../tut3_pymtl/regincr/RegIncr_extra_test.py::test_large PASSED
26 ../tut3_pymtl/regincr/RegIncr_test.py::test_basic PASSED
27
28 ================== 17 failed, 4 passed in 0.36 seconds ==================
```

Figure 14: `py.test` **Verbose Output –** Each line corresponds to one test case. Passing test cases are marked with `PASSED` and failing test cases are marked with `FAILED`.

Edit the new test script named `RegIncr_extra_test.py`. Add the code on lines 35–46 in Figure 13 which tests for overflow. Run this extra test script using `py.test` as follows:

```
% cd ${TUTROOT}/build
% py.test ../tut3_pymtl/regincr/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 13. The `py.test` framework can automatically discover test scripts in addition to automatically discovering the test cases within a test script. If the argument to `py.test` is a directory, then `py.test` will search that directory for any files ending in `_test.py` and assume that these files are test scripts. The `py.test` 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
% py.test ../tut3_pymtl/regincr -v
```

The verbose output should look similar to what is shown in Figure 14. 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
% py.test ../tut3_pymtl/regincr -k small
```

```
1  #-------------------------------------------------------------------------
2  # test_random
3  #-------------------------------------------------------------------------
4
5  import random
6
7  def test_random( dump_vcd ):
8
9    test_vector_table = [( 'in_', 'out*' )]
10   last_result = '?'
11   for i in xrange(20):
12     rand_value = Bits( 8, random.randint(0,0xff) )
13     test_vector_table.append( [ rand_value, last_result ] )
14     last_result = Bits( 8, rand_value + 1 )
15
16   run_test_vector_sim( RegIncr(), test_vector_table, dump_vcd )
```

Figure 15: 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.

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

```
% cd ${TUTROOT}/build
% py.test ../tut3_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 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
% py.test ../tut3_pymtl/regincr
% py.test ../tut3_pymtl/regincr/RegIncr2stage_test.py -v
% py.test ../tut3_pymtl/regincr/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.

4.7. 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 15 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
% py.test ../tut3_pymtl/regincr/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.

4.8. 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 16 shows a two-stage registered incrementer that uses structural composition to instantiate and connect two instances of a single-stage registered incrementer. Figure 17 shows the corresponding PyMTL model. Line 9 imports the child model that we will be reusing.

Lines 19–20 illustrate a simplified PyMTL syntax for specifying the type of the values that can be passed through the `in_` and `out` ports. If we use an integer `b`, then this is syntactic sugar for specifying that objects of type `Bits(b)` 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 `s.connect` method 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 `s.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.

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 18 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.

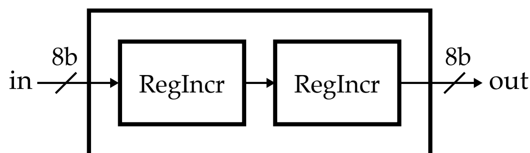

Figure 16: Block Diagram for Two-Stage Registered Incrementer – An eight-bit two-stage registered incrementer that reuses the registered incrementer in Figure 7 through structural composition.

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

from pymtl    import *
from RegIncr import RegIncr

class RegIncr2stage( Model ):

  # Constructor

  def __init__( s ):

    # Port-based interface

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

    # First stage

    s.reg_incr_0 = RegIncr()

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

    # Second stage

    s.reg_incr_1 = RegIncr()

    s.connect( s.reg_incr_0.out, s.reg_incr_1.in_ )
    s.connect( 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 17: Two-Stage Registered Incrementer – An eight-bit two-stage registered incrementer corresponding to Figure 16. This model is implemented using structural composition to instantiate and connect two instances of the single-stage register incrementer.

Edit the PyMTL source file named `RegIncr2stage.py`. Add lines 28-33 from Figure 17 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
% py.test ../tut3_pymtl/regincr/RegIncr2stage_test.py -v
% py.test ../tut3_pymtl/regincr/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:

```
% py.test ../tut3_pymtl/regincr/RegIncr2stage_test.py -k test_small -s
```

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

import random

from pymtl         import *
from pclib.test    import run_test_vector_sim
from RegIncr2stage import RegIncr2stage

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

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

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

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

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

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

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

def test_random( dump_vcd ):

  test_vector_table = [( 'in_', 'out*' )]
  last_result_0 = '?'
  last_result_1 = '?'
  for i in xrange(20):
    rand_value = Bits( 8, random.randint(0,0xff) )
    test_vector_table.append( [ rand_value, last_result_1 ] )
    last_result_1 = last_result_0
    last_result_0 = Bits( 8, rand_value + 2 )

  run_test_vector_sim( RegIncr2stage(), test_vector_table, dump_vcd )
```

Figure 18: Unit Test Script for Two-Stage Registered Incrementer – A unit test for the two-stage registered incrementer shown in Figure 17 that uses test vectors and the `py.test` unit testing framework.

```
1          reg_incr_0  reg_incr_1
2          ----------- -----------
3 cycle in  in reg out in reg  out out
4 ------------------------------------
5    2: 00 (00 (00) 01|01 (00) 01) 01
6    3: 03 (03 (00) 01|01 (01) 02) 02
7    4: 06 (06 (03) 04|04 (01) 02) 02
8    5: 00 (00 (06) 07|07 (04) 05) 05
9    6: 00 (00 (00) 01|01 (07) 08) 08
```

Figure 19: 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 19. 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
% py.test ../tut3_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 16. Verify your design by writing a test script that uses test vectors.

4.9. Parameterizing a Model with "Static" Elaboration

To facilitate model reuse and productive design-space exploration, we often want to implement parameterized models. Parameterized models take one or more parameters as constructor arguments, and then use these parameters when declaring the model's interface, defining the model's behavior in concurrent blocks, and/or structurally composing child models. A common example is to parameterize models 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 PyMTL framework includes a library of parameterized FL, CL, and RTL models called `pclib`. You can use the PyMTL GitHub repository (`http://github.com/cornell-brg/pymtl`) to browse what models are available in `pclib.rtl.arith`. Figure 20 shows a combinational incrementer from `pclib` that is parameterized by both the port bitwidth and the incrementer amount.

Figure 21 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 in the interface. This example illustrates how PyMTL enables powerful elaboration; we can use arbitrary Python code in a model's constructor to generate complex hardware based on the constructor argu-

```
class Incrementer( Model ):

  def __init__( s, nbits = 1, increment_amount = 1 ):

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

    s.increment_amount = increment_amount

    @s.combinational
    def comb_logic():
      s.out.value = s.in_ + s.increment_amount

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

Figure 20: Parameterized Incrementer from `pclib` – A combinational incrementer from `pclib` that is parameterized by both the port bitwidth and the incrementer amount.

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

from pymtl    import *
from RegIncr import RegIncr

class RegIncrNstage( Model ):

  # Constructor

  def __init__( s, nstages=2 ):

    # Port-based interface

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

    # Instantiate the registered incrementers

    s.reg_incrs = [ RegIncr() for x in xrange(nstages) ]

    # Connect input port to first reg_incr in chain

    s.connect( s.in_, s.reg_incrs[0].in_ )

    # Connect reg_incr in chain

    for i in xrange( nstages - 1 ):
      s.connect( s.reg_incrs[i].out, s.reg_incrs[i+1].in_ )

    # Connect last reg_incr in chain to output port

    s.connect( s.reg_incrs[-1].out, s.out )

  # Line Tracing

  def line_trace( s ):
    return "{} ({}) {}".format(
      s.in_,
      '|'.join([ str(reg_incr.out) for reg_incr in s.reg_incrs ]),
      s.out
    )
```

Figure 21: N-Stage Registered Incrementer – A parameterized registered incrementer where the number of stages is specified as an argument to the constructor.

ments. In traditional hardware description languages, this process is often called static elaboration since this phase happens at compile or synthesis time. In PyMTL, 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 PyMTL 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 `py.test` framework includes sophisticated support for parameterized testing that can simplify verifying highly parameterized models. Figure 22 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 18–28 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 22 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 `py.test` framework will automatically generate a set of test cases for each set of argument values.

On lines 34–51, 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 PyMTL 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 47–51 includes a `test_params` argument that will contain the test parameters corresponding to one row of the test case table. On lines 48–49, we read these test parameters, and then on lines 50–51 we use the `run_test_vector_sim` and the `mk_test_vector_table` helper functions to actually run a test.

On lines 57–60, 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 `py.test` 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

from pymtl         import *
from pclib.test    import run_test_vector_sim, mk_test_case_table
from RegIncrNstage import RegIncrNstage

#-------------------------------------------------------------------------
# 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( Bits( 8, input_ + nstages, trunc=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, dump_vcd ):
  nstages = test_params.nstages
  inputs  = test_params.inputs
  run_test_vector_sim( RegIncrNstage( nstages ),
    mk_test_vector_table( nstages, inputs ), dump_vcd )

#-------------------------------------------------------------------------
# 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, dump_vcd ):
  run_test_vector_sim( RegIncrNstage( nstages=n ),
    mk_test_vector_table( n, sample(range(0xff),20) ), dump_vcd )
```

Figure 22: Unit Test Script for Parameterized Registered Incrementer – A unit test for the parameterized registered incrementer shown in Figure 21.

```
1  ========================= test session starts ==========================
2  platform darwin -- Python 2.7.5 -- py-1.4.26 -- pytest-2.6.4
3  plugins: xdist
4  collected 14 items
5
6  ../tut3_pymtl/regincr/RegIncrNstage_test.py::test[2stage_small] PASSED
7  ../tut3_pymtl/regincr/RegIncrNstage_test.py::test[2stage_large] PASSED
8  ../tut3_pymtl/regincr/RegIncrNstage_test.py::test[2stage_overflow] PASSED
9  ../tut3_pymtl/regincr/RegIncrNstage_test.py::test[2stage_random] PASSED
10 ../tut3_pymtl/regincr/RegIncrNstage_test.py::test[3stage_small] PASSED
11 ../tut3_pymtl/regincr/RegIncrNstage_test.py::test[3stage_large] PASSED
12 ../tut3_pymtl/regincr/RegIncrNstage_test.py::test[3stage_overflow] PASSED
13 ../tut3_pymtl/regincr/RegIncrNstage_test.py::test[3stage_random] PASSED
14 ../tut3_pymtl/regincr/RegIncrNstage_test.py::test_random[1] PASSED
15 ../tut3_pymtl/regincr/RegIncrNstage_test.py::test_random[2] PASSED
16 ../tut3_pymtl/regincr/RegIncrNstage_test.py::test_random[3] PASSED
17 ../tut3_pymtl/regincr/RegIncrNstage_test.py::test_random[4] PASSED
18 ../tut3_pymtl/regincr/RegIncrNstage_test.py::test_random[5] PASSED
19 ../tut3_pymtl/regincr/RegIncrNstage_test.py::test_random[6] PASSED
20
21 ====================== 14 passed in 0.17 seconds =======================
```

Figure 23: `py.test` **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 PyMTL source file named `RegIncrNstage.py`. Add the code on lines 28–31 from Figure 21 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
% py.test ../tut3_pymtl/regincr/RegIncrNstage_test.py -v
```

The output should look similar to what is shown in Figure 23. Notice how the `py.test` framework names the generated test cases. When using a test case table, the `py.test` 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 `py.test` 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 `py.test` 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
% py.test ../tut3_pymtl/regincr/RegIncrNstage_test.py -k 3stage -s
```

★ *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.

4.10. Packaging a Collection of Models

We group related models into a single subdirectory (sometimes called a "subproject") within a PyMTL project. Packaging is the process of making a subproject available for other subprojects to use via the

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

from RegIncr       import RegIncr
from RegIncr2stage import RegIncr2stage
from RegIncrNstage import RegIncrNstage
```

Figure 24: 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 pymtl import *
>>> from tut3_pymtl.regincr import RegIncr
>>> model = RegIncr()
>>> model.elaborate()
>>> sim = SimulationTool( model )
>>> sim.reset()
>>> model.in_.value = 0x24
>>> sim.cycle()
>>> model.out
Bits( 8, 0x25 )
```

Figure 25: 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.name for x in model.get_ports() ]
['reset', 'in_', 'clk', 'out']
>>> [ x.name for x in model.get_wires() ]
['reg_out']
```

Figure 26: Importing a PyMTL Package from the Build Directory

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 PyMTL 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 `tut3_pymtl` subdirectory.

Figure 24 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 25 shows an example session in the Python interpreter that illustrates how to import models from the `regincr` package and then use the `SimulationTool` 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 `tut3_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 26 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 26 also illustrates how the PyMTL framework provides an interface for inspecting elaborated models. The `get_ports` method will return a list of input/output ports for an elaborated model. There are similar methods for inspecting a model's wires, child models, connections, and concurrent blocks. This interface is often used when implementing new PyMTL tools, but can also be potentially useful when implementing highly parameterized models.

5. Sort Unit

The previous section introduced the key PyMTL concepts and primitives that we will use to implement more complex FL, CL, and RTL models including: using the `Model` base class to define PyMTL models; declaring the port-based interfaces using the `InPort` and `OutPort` classes; declaring internal wires using the `Wire` class; declaring `s.tick` concurrent blocks to model logic that executes on every rising clock edge; declaring `s.combinational` concurrent blocks to model combinational logic that executes one or more times within a clock cycle; using structural composition to connect child models; and creating parameterized models. 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 PyMTL translation tool to generate Verilog from an RTL model. Most of the code for this section is provided for you in the `tut3_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 27 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 28 shows how to implement an FL model for the sort unit in PyMTL. On lines 16 and 19, we use Python list comprehensions to create lists of four input and output ports. On lines 31 and 35, 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 `s.tick` concurrent block on lines 21–25 defines the actual functional-level behavior. PyMTL provides specialized decorators for FL, CL, and RTL modeling, so we use the `s.tick_fl` decorator instead of the generic `s.tick` decorator used in the previous section. You should always use the more specialized decorators instead of the generic `s.tick` decorator to better capture your intent.

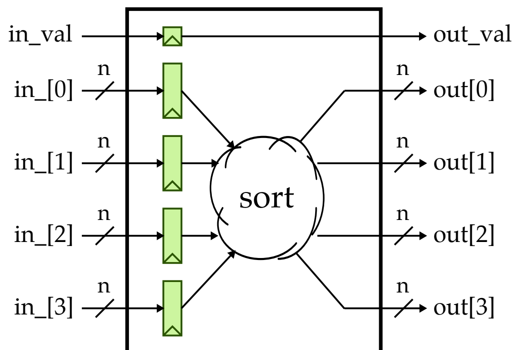

Figure 27: Cloud Diagram for Sort Unit FL Model – Cloud diagrams use "clouds" to abstractly represent logic without worry about the actual implementation details. The sort unit FL model takes four input values and sorts them such that the `out[0]` port has the smallest value and the `out[3]` port has 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 pymtl import *

class SortUnitFL( Model ):

  # Constructor

  def __init__( s, nbits=8 ):

    s.in_val  = InPort(1)
    s.in_     = [ InPort  (nbits) for _ in range(4) ]

    s.out_val = OutPort(1)
    s.out     = [ OutPort (nbits) for _ in range(4) ]

    @s.tick_fl
    def block():
      s.out_val.next = s.in_val
      for i, v in enumerate( sorted( s.in_ ) ):
        s.out[i].next = 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 "{}|{}".format( in_str, out_str )
```

Figure 28: Sort Unit FL Model – FL model of four-element sort unit corresponding to Figure 27.

The more specialized decorators also provide additional functionality that is appropriate for each abstraction level. We will still use the term `s.tick` concurrent block to generically refer to any of these types of concurrent blocks. The `s.tick` concurrent block in our sort unit FL model uses the standard Python `sorted` function and then 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.

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 PyMTL semantics of `s.tick` concurrent blocks, and this is why we show input registers in the cloud diagram in Figure 27. Although it is also possible to implement FL models using `s.combinational` concurrent blocks, we have found using `s.tick` concurrent blocks to be significantly easier. Using `s.combinational` 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.

We do not explicitly handle resetting the valid bit, but we instead rely on the PyMTL 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 PyMTL model is in `SortUnitFL.py` and the corresponding test script is in `SortUnitFL_test.py`. This test script uses test vector tables similar in spirit to the unit testing for the registered incrementer in Figure 18. The test script 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
% py.test ../tut3_pymtl/sort/SortUnitFL_test.py -v
% py.test ../tut3_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 29 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.

Figure 30 shows how to implement a CL model for the sort unit in PyMTL. 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 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.

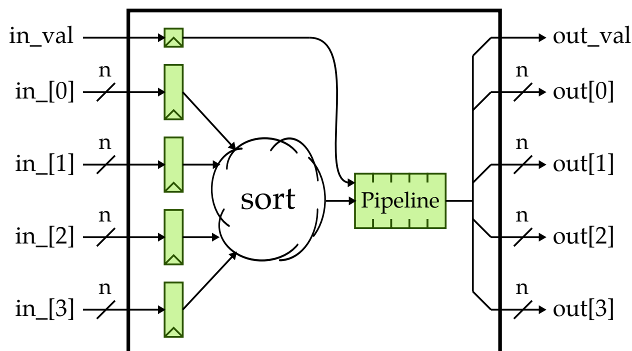

Figure 29: Cloud Diagram for Sort Unit CL Model – The CL model completely sorts the input values in the first cycle, and then uses a pipeline object to model the pipeline latency.

```
 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 pymtl       import *
10
11 class SortUnitCL( Model ):
12
13   # Constructor
14
15   def __init__( s, nbits=8, nstages=3 ):
16
17     s.in_val  = InPort (1)
18     s.in_     = [ InPort  (nbits) for _ in range(4) ]
19
20     s.out_val = OutPort(1)
21     s.out     = [ OutPort (nbits) for _ in range(4) ]
22
23     s.pipe    = deque( [[0,0,0,0,0]]*(nstages-1) )
24
25     @s.tick_cl
26     def block():
27       s.pipe.append( deepcopy( [s.in_val] + sorted(s.in_) ) )
28       data = s.pipe.popleft()
29       s.out_val.next = data[0]
30       for i, v in enumerate( data[1:] ):
31         s.out[i].next = v
32
33   # Line tracing
34
35   def line_trace( s ):
36
37     in_str = '{' + ','.join(map(str,s.in_)) + '}'
38     if not s.in_val:
39       in_str = ' '*len(in_str)
40
41     out_str = '{' + ','.join(map(str,s.out)) + '}'
42     if not s.out_val:
43       out_str = ' '*len(out_str)
44
45     return "{}|{}".format( in_str, out_str )
```

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

The `s.tick_cl` concurrent block on lines 25–31 defines the actual functional-level behavior. We first sort the input values using the standard Python `sorted` function and append the corresponding sorted list of four values along with the input valid bit to the back of the `deque` (line 27). We then pop the next list of four values from the front of the `deque` (line 28), write the valid bit to the `out_val` port (line 29), and write the sorted list to the `out` ports (lines 30–31). Notice how line 23 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 27 we will append the list of sorted values to the `deque`, and on line 28 we will immediately pop this same list of sorted values from the `deque`. In

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

Figure 31: 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.

this case, the `s.tick` 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 27. 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 PyMTL 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 22. 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
% py.test ../tut3_pymtl/sort/SortUnitCL_test.py -v
% py.test ../tut3_pymtl/sort/SortUnitCL_test.py -k 3stage_stream -s
```

Figure 31 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`).

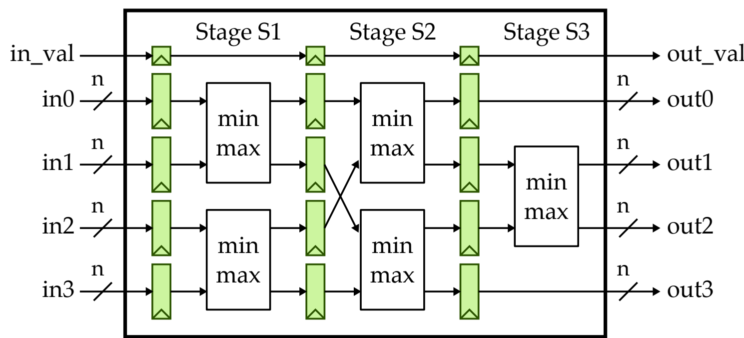

Figure 32: Block Diagram for Sort Unit RTL Model – The RTL model implements a three-stage pipelined, bitonic sorting network.

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 32 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 33 shows how to implement a flat RTL model for the sort unit in PyMTL. 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 `s.tick_rtl` concurrent blocks) from the combinational logic (modeled with `s.combinational` 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 PyMTL framework to reset state. Line 30 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 31–32 illustrate a loop that iterates over the `in_` ports to model the input registers. Lines 40–59 correspond to the first stage in Figure 32 with two min/max units.

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

from pymtl import *

class SortUnitFlatRTL( Model ):

  def __init__( s, nbits=8 ):

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

    s.in_val  = InPort (1)
    s.in_     = [ InPort  (nbits) for _ in range(4) ]

    s.out_val = OutPort(1)
    s.out     = [ OutPort (nbits) for _ in range(4) ]

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

    s.val_S1 = Wire(1)
    s.elm_S1 = [ Wire(nbits) for _ in range(4) ]

    @s.tick_rtl
    def pipereg_S0S1():
      s.val_S1.next  = s.in_val if ~s.reset else 0
      for i in xrange(4):
        s.elm_S1[i].next = s.in_[i]

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

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

    @s.combinational
    def stage_S1():

      # Sort elements 0 and 1

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

      # Sort elements 2 and 3

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

    ...
```

Figure 33: Sort Unit Flat RTL Model – RTL model of four-element sort unit corresponding to Figure 32. For simplicity only the interface and first pipeline stage are shown.

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

Figure 34: 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.

The PyMTL 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
% py.test ../tut3_pymtl/sort/SortUnitFlatRTL_test.py -v
% py.test ../tut3_pymtl/sort/SortUnitFlatRTL_test.py -k test_basic -s
```

The line trace for the sort unit RTL model is shown in Figure 34. On cycle 3, there is a valid set of four input values available on the input ports, and on cycle 4, 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 4, 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 5, pipeline stage S2 swaps elements 0 and 2, and also swaps elements 1 and 3. During cycle 6, 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 31 and the sort unit RTL model in Figure 34. 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 3, and a sorted set of four values is produced by the sort unit model on cycle 6. 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 35 shows how to implement a structural RTL model for the sort unit in PyMTL. 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 im-

```
1  #=========================================================================
2  # SortUnitStructRTL
3  #=========================================================================
4
5  from pymtl          import *
6  from pclib.rtl.regs import Reg, RegRst
7  from MinMaxUnit     import MinMaxUnit
8
9  class SortUnitStructRTL( Model ):
10
11   def __init__( s, nbits=8 ):
12
13     #---------------------------------------------------------------------
14     # Interface
15     #---------------------------------------------------------------------
16
17     s.in_val  = InPort (1)
18     s.in_     = [ InPort  (nbits) for _ in range(4) ]
19
20     s.out_val = OutPort(1)
21     s.out     = [ OutPort (nbits) for _ in range(4) ]
22
23     #---------------------------------------------------------------------
24     # Stage S0->S1 pipeline registers
25     #---------------------------------------------------------------------
26
27     s.val_S0S1 = RegRst(1)
28     s.elm_S0S1 = [ Reg(nbits) for _ in range(4) ]
29
30     s.connect( s.in_val, s.val_S0S1.in_ )
31     for i in xrange(4):
32       s.connect( s.in_[i], s.elm_S0S1[i].in_ )
33
34     #---------------------------------------------------------------------
35     # Stage S1 combinational logic
36     #---------------------------------------------------------------------
37
38     s.minmax0_S1 = MinMax(nbits)
39
40     s.connect( s.elm_S0S1[0].out, s.minmax0_S1.in0 )
41     s.connect( s.elm_S0S1[1].out, s.minmax0_S1.in1 )
42
43     s.minmax1_S1 = MinMax(nbits)
44
45     s.connect( s.elm_S0S1[2].out, s.minmax1_S1.in0 )
46     s.connect( s.elm_S0S1[3].out, s.minmax1_S1.in1 )
47
48     ...
```

Figure 35: Sort Unit Structural RTL Model – RTL model of four-element sort unit corresponding to Figure 32. For simplicity only the interface and first pipeline stage are shown.

plementation strategy, we still cleanly separate the sequential child models from the combinational child models. We still use comments and explicit suffixes to make it clear what pipeline stage we are modeling.

Notice on lines 27–28 we are using register models from `pclib`. On line 6, we import the `Reg` (simple positive-edge triggered register) and `RegRst` (positive-edge triggered register with reset) models. Notice our use of a loop to connect the `in_` ports in the interface to the `in_` ports in the `Reg` model. As shown on lines 38–46, we usually instantiate a child model, and then we use `s.connect` statements to implement structural composition. There is no need to declare intermediate wires; we can directly connect ports between two different models.

The PyMTL 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 `py.test` framework.

★ *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 `py.test`. 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
% py.test ../tut3_pymtl/sort/SortUnitStructRTL_test.py -v
% py.test ../tut3_pymtl/sort/SortUnitStructRTL_test.py -k test_basic -s
```

The line trace for the sort unit structural RTL model should be the same as in Figure 34, 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 36. 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

```
opts = parse_cmdline()

# Create input datasets

ninputs = 100
inputs  = []

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

...

# Instantiate and elaborate the design

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

model = model_impl_dict[ opts.impl ]()

dump_vcd = ""
if opts.dump_vcd:
  dump_vcd = "sort-" + opts.impl + "-" + opts.input + ".vcd"

model.vcd_file = dump_vcd

model.elaborate()
sim = SimulationTool( 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.value = True
    for i,v in enumerate( inputs.pop() ):
      model.in_[i].value = v

  else:
    model.in_val.value = False
    for i in xrange(4):
      model.in_[i].value = 0

  if opts.trace:
    sim.print_line_trace()

  sim.cycle()

# 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 36: 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.

on the `--impl` command line option. The simulator will conditionally generate waveforms based on the `--dump-vcd` command line option. The main simulator loop on lines 37–55 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. Lines 52–53 turn on line tracing based on the `--trace` command line option. 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
% ../tut3_pymtl/sort/sort-sim --stats --impl cl
% ../tut3_pymtl/sort/sort-sim --stats --impl rtl-flat
% ../tut3_pymtl/sort/sort-sim --stats --impl rtl-struct
```

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
% ../tut3_pymtl/sort/sort-sim --stats --impl cl --input random
% ../tut3_pymtl/sort/sort-sim --stats --impl cl --input sorted-fwd
% ../tut3_pymtl/sort/sort-sim --stats --impl cl --input sorted-rev
```

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

```
% cd ${TUTROOT}/build
% ../tut3_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,

```
1  % cd ${TUTROOT}/build
2  % env PYTHONPATH=".." python
3  >>> from pymtl import *
4  >>> from tut3_pymtl.sort import SortUnitFlatRTL
5  >>> model = SortUnitFlatRTL()
6  >>> model = TranslationTool( model )
7  % ls
8  SortUnitFlatRTL_0x4b8e51bd8055176a.v
9  SortUnitFlatRTL_0x4b8e51bd8055176a_v.cpp
10 SortUnitFlatRTL_0x4b8e51bd8055176a_v.py
11 SortUnitFlatRTL_0x4b8e51bd8055176a_v.pyc
12 libSortUnitFlatRTL_0x4b8e51bd8055176a_v.so
13 obj_dir_SortUnitFlatRTL_0x4b8e51bd8055176a
```

Figure 37: Translating an RTL Model into Verilog – The `TranslationTool` translates an RTL model into Verilog, but also uses the Verilator tool and various generated wrappers to creates a new PyMTL model that internally contains its own cycle-accurate simulator for the translated Verilog.

energy, and timing. PyMTL currently supports translating an RTL model into Verilog, although the framework's use of a clean model/tool split can enable adding translation tools for other HDLs in the future.

Figure 37 shows an example session in the Python interpreter that illustrates how to use the translation tool from the PyMTL 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_0x4b8e51bd8055176a.v` to see the Verilog generated by the translation tool. The suffix `0x4b8e51bd8055176a` corresponds to a hash of the design parameters; this ensures that the generated Verilog module name is unique across different instantiations of the same parameterized model. Notice that the translation tool preserves the model hierarchy, unrolls lists, and uses relatively readable name mangling from PyMTL to Verilog names. `s.tick_rtl` concurrent blocks are translated into Verilog `always @( posedge clk )` concurrent blocks, and `s.combinational` concurrent blocks are translated into Verilog `always @(*)` concurrent blocks. Also notice that for each concurrent block, the translation tool includes the corresponding PyMTL code as a comment directly above the generated Verilog. This can be useful when debugging incorrect translations.

The translation tool actually does far more than just translate RTL models into Verilog. The translation tool will: (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 PyMTL wrapper around the shared library. Essentially, this means the translation tool creates a new PyMTL 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 PyMTL wrappers. This powerful feature enables us to seamlessly use the exact same test scripts to verify the functionality of the translated Verilog.

Figure 38 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 11, except of course our sort unit requires many more input and output values. 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`. On lines 17–18, we use the translation tool to first translate the sort unit into Verilog and then return a new PyMTL model that contains the translated Verilog. We can now test both the PyMTL RTL and the translated Verilog as follows:

```
% cd ${TUTROOT}/build
% py.test ../tut3_pymtl/sort/SortUnitFlatRTL_v_test.py --dump-vcd
% mv tut3_pymtl.sort.SortUnitFlatRTL_v_test.test_verilate.vcd sort-pymtl.vcd
% py.test ../tut3_pymtl/sort/SortUnitFlatRTL_v_test.py --dump-vcd --test-verilog
```

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

from pymtl           import *
from SortUnitFlatRTL import SortUnitFlatRTL

def test_verilate( dump_vcd, test_verilog ):

  # Conflat the model

  model = SortUnitFlatRTL()
  model.vcd_file = dump_vcd

  # Translate the model into Verilog

  if test_verilog:
    model = TranslationTool( model )

  # Elaborate the model

  model.elaborate()

  # Create and reset simulator

  sim = SimulationTool( model )
  sim.reset()
  print ""

  # Helper function

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

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

    sim.eval_combinational()
    sim.print_line_trace()

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

    sim.cycle()

  # 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 38: 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`.

```
% ls *SortUnitFlatRTL_v_test.*.vcd
```

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

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

However, sometimes the translation tool 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 *
% py.test ../tut3_pymtl/sort/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_sim` helper function accepts `test_verilog` as an argument. This enables us to test the translated Verilog on all of our tests as follows:

```
% cd ${TUTROOT}/build
% py.test ../tut3_pymtl/sort -v --test-verilog
```

You should see that `py.test` 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.

```
% ../tut3_pymtl/sort/sort-sim --impl rtl-flat --input random     --translate --dump-vcd
% ../tut3_pymtl/sort/sort-sim --impl rtl-flat --input sorted-fwd --translate --dump-vcd
% ../tut3_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 val/rdy interfaces, unit testing with test sources/sinks, and using a control/datapath split to implement RTL models. The code for this section is provided for you in the `tut3_pymtl/gcd` subdirectory. 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 `tut3_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 39 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 val/rdy 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 39, 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

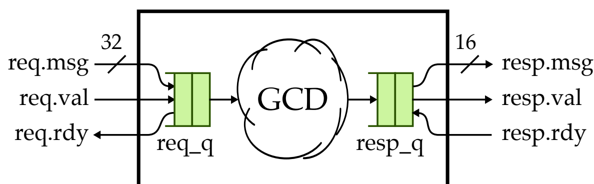

Figure 39: Cloud Diagram for GCD Unit FL Model – Input and output use latency-insensitive val/rdy interfaces; input/output queue interface adapters simplify interacting with these interfaces. The input message includes two 16-bit operands; output message is an 16-bit result.

```
 1  #=========================================================================
 2  # GCD Unit FL Model
 3  #=========================================================================
 4
 5  from fractions  import gcd
 6
 7  from pymtl      import *
 8  from pclib.ifcs import InValRdyBundle, OutValRdyBundle
 9  from pclib.fl   import InValRdyQueueAdapter, OutValRdyQueueAdapter
10
11  class GcdUnitFL( Model ):
12
13    # Constructor
14
15    def __init__( s ):
16
17      # Interface
18
19      s.req    = InValRdyBundle  (32)
20      s.resp   = OutValRdyBundle (16)
21
22      # Adapters
23
24      s.req_q  = InValRdyQueueAdapter  ( s.req  )
25      s.resp_q = OutValRdyQueueAdapter ( s.resp )
26
27      # Concurrent block
28
29      @s.tick_fl
30      def block():
31        req_msg = s.req_q.popleft()
32        result = gcd( req_msg[0:16], req_msg[16:32] )
33        s.resp_q.append( result )
34
35    # Line tracing
36
37    def line_trace( s ):
38      req_msg_str = "{}:{}".format( s.req.msg[0:16], s.req.msg[16:32] )
39      return "{}(){}".format(
40        valrdy_to_str( req_msg_str, s.req.val, s.req.rdy ),
41        s.resp
42      )
```

Figure 40: Gcd Unit FL Model – FL model of greatest-common divisor unit corresponding to Figure 39.

to make the valid signal depend on the ready signal. As long as you adhere to this policy, composing modules via the val/rdy interface should not cause significant timing issues.

Based on the discussion so far, the benefit of a latency-insensitive 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.

In Figure 39, we can see that we often use input/output queues to simplify designing FL models that interact with val/rdy interfaces. Figure 40 shows how to implement an FL model for the GCD unit in PyMTL. The actual work of the FL model takes place on line 32. We use the `gcd` function from the standard Python `fractions` module to calculate the GCD of the two input operands. This exam-

```
1  #=========================================================================
2  # ValRdyBundle.py
3  #=========================================================================
4  # Defines a PortBundle for the val/rdy interface.
5
6  from pymtl  import *
7  from valrdy import valrdy_to_str
8
9  class ValRdyBundle( PortBundle ):
10
11   def __init__( self, nbits ):
12     self.msg = InPort ( nbits )
13     self.val = InPort ( 1 )
14     self.rdy = OutPort( 1 )
15
16   def to_str( self, msg=None ):
17     msg = self.msg if None else msg
18     return valrdy_to_str( msg, self.val, self.rdy )
19
20   def __str__( self ):
21     return valrdy_to_str( self.msg, self.val, self.rdy )
22
23 # Create InValRdyBundle and OutValRdyBundle
24
25 InValRdyBundle, OutValRdyBundle = create_PortBundles( ValRdyBundle )
```

Figure 41: Val/Rdy Port Bundle from `pclib` – A parameterized port bundle that groups together the valid, ready, and message ports.

ple illustrates a two important new features of the PyMTL framework: port bundles and interface adapters.

Lines 19–20 of Figure 40 use port bundles instead of ports as the interface for our GCD unit. A port bundle is simply a collection of logically related ports (potentially in different directions), which can then be connected in a single statement. For our GCD unit, we are using the `ValRdyBundle` from `pclib.ifcs`. This port bundle groups together the valid, ready, and message ports. Figure 41 shows how the port bundle is defined in `pclib`. A port bundle is just a Python class that inherits from the `PortBundle` base class provided by the PyMTL framework. In the constructor, we define the ports that make up the port bundle (lines 12–14). We also define methods for converting the port bundle to a string for simplified line tracing. One line 25, we use the `create_PortBundles` function from the PyMTL framework to create two new port bundle classes: `InValRdyBundle` has input valid/message ports and an output ready port, while `OutValRdyBundle` as output valid/message ports and an input ready port.

Lines 24–25 of Figure 40 instantiate two interface adapters provided by the PyMTL framework. Interface adapters take one or more ports (or port bundles) as constructor arguments, and then enable the logic within the model to interact with these ports through methods. In this example, we are using `ValRdyQueueAdapter` objects from `pclib.fl`. An `InValRdyQueueAdapter` accepts an `InValRdyBundle` and provides a standard Python `popleft` method for the FL model to use. An `OutValRdyQueueAdapter` accepts an `OutValRdyBundle` and provides a standard Python `append` method for the FL model to use. We can see the `s.tick` concurrent block making use of these methods to pop a request message from the input interface (line 31) and append the response message on the output interface (line 33). These queue adapters significantly simplify implementing FL models, since we no longer need to explicitly manage the val/rdy interface. The framework actually uses a sophisticated implementation to enable an `s.tick` concurrent block to be "paused" if the input interface is not valid or the output interface is not ready.

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

class TestHarness (Model):

  def __init__( s, GcdUnit, src_msgs, sink_msgs,
                   src_delay, sink_delay,
                   dump_vcd=False, test_verilog=False ):

    # Instantiate models

    s.src  = TestSource ( 32, src_msgs,  src_delay  )
    s.gcd  = GcdUnit    ()
    s.sink = TestSink   ( 16, sink_msgs, sink_delay )

    # Dump VCD

    if dump_vcd:
      s.gcd.vcd_file = dump_vcd

    # Translation

    if test_verilog:
      s.gcd = get_verilated( s.gcd )

    # Connect

    s.connect( s.src.out,  s.gcd.req  )
    s.connect( s.gcd.resp, s.sink.in_ )

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

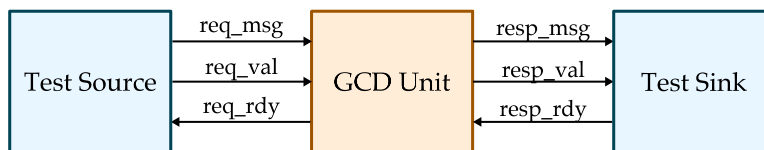

Figure 43: Verifying GCD Using Test Sources and Sinks – Parameterized test sources send a stream of messages over a val/rdy interface, and parameterized test sinks receive a stream of messages over a val/rdy interface and compare each message to a previously specified reference message.

The PyMTL 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 val/rdy interface is that it enables us to use a common framework for sending messages into the device-under-test (DUT) and then verifying that the correct messages come out of the DUT. `pclib.test` includes the `TestSource` and `TestSink` models for this purpose. Figure 42 illustrates the test harness included in the GCD unit test script. We instantiate a test source and attach it to the GCD unit's request val/rdy interface, and then we instantiate a test sink and attach it to the GCD unit's response val/rdy interface. Figure 43

```
1  cycle src        A    B     out    sink
2  ---------------------------------------
3    19: 0004001c > 001c:0004().    > .
4    20:          >          ()#    > #
5    21:          >          ()#    > #
6    22: #        > #        ()#    > #
7    23: #        > #        ()#    > #
8    24: #        > #        ()#    > #
9    25: 00990096 > 0096:0099()0004 > 0004
10   26:          >          ()#    > #
11   27:          >          ()#    > #
12   28: 009c00b4 > 00b4:009c()0003 > 0003
13   29: .        > .        ()000c > 000c
14   30: 00a00060 > 0060:00a0().    > .
15   31: #        > #        ()#    > #
16   32: 005400a4 > 00a4:0054()0020 > 0020
17   33: #        > #        ()#    > #
18   34: 00ab0059 > 0059:00ab()0004 > 0004
```

Figure 44: 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.

illustrates the overall connectivity in the test harness. Notice how port bundles enable us to connect three ports with a single connect statement (lines 29–30). The test source includes the ability to randomly delay messages going into the DUT and the test sink includes the ability to randomly apply back-pressure to the DUT. By using various combinations of these random 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 random 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
% py.test ../tut3_pymtl/gcd/test/GcdUnitFL_test.py -k basic_0x0 -s
```

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

```
% cd ${TUTROOT}/build
% py.test ../tut3_pymtl/gcd/test/GcdUnitFL_test.py -k basic -s
```

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

```
% cd ${TUTROOT}/build
% py.test ../tut3_pymtl/gcd/test/GcdUnitFL_test.py -k random -s
```

Figure 44 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 19 and although the result is valid on cycle 20 the test sink is not ready until cycle 25 to accept the result. On cycles 20–21 the test source does not have a new message to send to the GCD unit. On cycle 22 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 25 the test sink is ready and the GCD unit is able to send the result and accept a new input. The GCD unit takes a single cycle when there is no back pressure; we can see this on cycle 28–29.

★ *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 random delays, and the second new test case should have random 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 29 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 46 shows an excerpt from the CL model for the GCD unit. Lines 5–15 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 port bundles and interface adapters.

The interface adapters on lines 32–33 are different from the ones we used in the GCD unit FL model. These queue adapters still accept val/rdy port bundles, but they are meant for CL instead of FL modeling. We must explicitly tick them once a cycle (lines 47–48). We can use the `empty` method to see if an input queue is empty, and (if not empty) we can use the `deq` method to dequeue a message

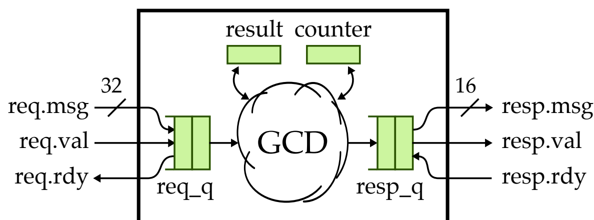

Figure 45: Cloud Diagram for GCD Unit CL Model – CL model uses input/output queue adapters and extra state to create a cycle-level timing model.

```
#=========================================================================
# 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)

#=========================================================================
# GcdUnitCL
#=========================================================================

class GcdUnitCL( Model ):

  def __init__( s ):

    # Interface

    s.req    = InValRdyBundle  (32)
    s.resp   = OutValRdyBundle (16)

    # Adapters

    s.req_q  = InValRdyQueueAdapter  ( s.req  )
    s.resp_q = OutValRdyQueueAdapter ( s.resp )

    # Member variables

    s.result  = 0
    s.counter = 0

    # Concurrent block

    @s.tick_cl
    def block():

      # Tick the queue adapters

      s.req_q.xtick()
      s.resp_q.xtick()

      # Handle delay to model the gcd unit latency

      if s.counter > 0:
        s.counter -= 1
        if s.counter == 0:
          s.resp_q.enq( s.result )

      # If we have a new message and the output queue is not full

      elif not s.req_q.empty() and not s.resp_q.full():
        req_msg = s.req_q.deq()
        s.result,s.counter = gcd( req_msg[0:16], req_msg[16:32] )
```

Figure 46: Excerpt from Gcd Unit CL Model – CL model of greatest-common divisor unit corresponding to Figure 45.

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

import pytest

from pymtl      import *
from pclib.test import run_sim
from tut3_pymtl.gcd.GcdUnitCL  import gcd, GcdUnitCL

# Reuse tests from FL model

from GcdUnitFL_test import TestHarness
from GcdUnitFL_test import basic_msgs, random_msgs, test_case_table

#-------------------------------------------------------------------------
# test_gcd
#-------------------------------------------------------------------------

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

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

@pytest.mark.parametrize( **test_case_table )
def test( test_params, dump_vcd ):
  run_sim( TestHarness( GcdUnitCL,
                        test_params.msgs[::2], test_params.msgs[1::2],
                        test_params.src_delay, test_params.sink_delay ),
           dump_vcd )
```

Figure 47: 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.

from the input queue. We can use the `full` method to see if an output queue is full, and (if not full) we can use the `enq` method to enqueue a message onto the output queue. These queue adapters significantly simplify implementing CL models, since we no longer need to explicitly manage the val/rdy interface. However, these queue adapters do 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.

Figure 47 shows the unit test script for our GCD unit CL model. Lines 20–30 use directed testing for just the algorithm and the associated timing model. Line 9 imports the GCD CL unit by its absolute package because it is not in the same subdirectory as the test file. Lines 13–14 import the test harness, messages, 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 36–41. 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 val/rdy interfaces combined

```
1 cycle src        A    B     out    sink
2 ---------------------------------------
3    2:          >            ()     >
4    3: 000f0005 > 0005:000f()     >
5    4: 00030009 > 0009:0003()     >
6    5: #        > #          ()     >
7    6: #        > #          ()     >
8    7: #        > #          ()     >
9    8: #        > #          ()     >
10   9: #        > #          ()     >
11  10: #        > #          ()0005 > 0005
12  11: 00000000 > 0000:0000().    > .
13  12: #        > #          ()     >
14  13: #        > #          ()     >
15  14: #        > #          ()     >
16  15: #        > #          ()     >
17  16: #        > #          ()0003 > 0003
18  17: 001b000f > 000f:001b().    > .
19  18: #        > #          ()#    > #
20  19: #        > #          ()#    > #
21  20: #        > #          ()#    > #
22  21: #        > #          ()#    > #
23  22: #        > #          ()0000 > 0000
24  23: 00150031 > 0031:0015().    > .
25  24: #        > #          ().    > .
```

Figure 48: 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.

with the flexibility of the `py.test` framework enable reusing tests across different models. This is an incredibly useful feature and significantly simplifies test-driven development.

The PyMTL 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 random delays in the test sink like this:

```
% cd ${TUTROOT}/build
% py.test ../tut3_pymtl/gcd/test/GcdUnitCL_test.py -v
% py.test ../tut3_pymtl/gcd/test/GcdUnitCL_test.py -sv -k basic_0x5
```

Figure 48 shows the beginning of the line trace for the basic test case. The first GCD request enters the GCD unit on cycle 3 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 4 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 16. The third GCD request stalls until cycle 11 when it can enter the GCD unit. On cycle 18, 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 random delay). The GCD unit must wait until the test sink is ready on cycle 22.

★ *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 models, 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

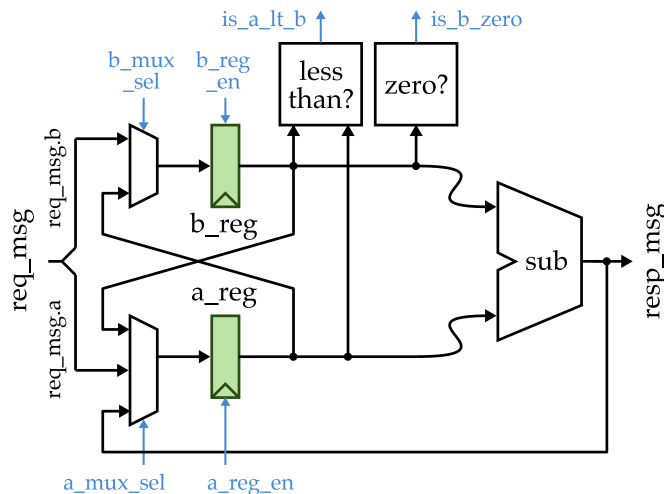

Figure 49: Datapath Diagram for GCD – Datapath includes two state registers and required muxing and arithmetic units to iteratively implement Euclid's algorithm.

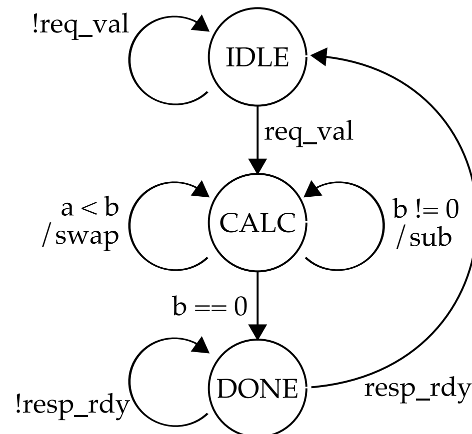

Figure 50: FSM Diagram for GCD – A hybrid Moore/Mealy FSM for controlling the datapath in Figure 49. Mealy transitions in the calc state determine whether to swap or subtract.

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 49 illustrates the datapath for the GCD unit and Figure 50 illustrates the corresponding finite-state-machine (FSM) control unit. The PyMTL source for the datapath, control unit, and the top-level module which composes the datapath and control unit is in `GcdUnitRTL.py`.

Figure 51 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 49. We leverage several modules from `pclib` (e.g., `Mux`, `RegEn`). You should use a similar structural approach when building your own datapaths for this course. Lines 41–46 illustrate using the `s.connect_pairs` method to create a different style for structural composition well-suited for implementing datapaths. The `s.connect_pairs` method takes a list of ports that need to be connected. The first port is connected to the second port, the third is connected to the fourth and so on. 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 `s.combinational` concurrent block for the state transitions, and an `s.combinational` 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. We use the `s.connect_auto` method to connect the control and status signals between the datapath and control unit. The `s.connect_auto` method will inspect the port lists for the given models and connect ports with the same name. For example, the `a_mux_sel` port in the datapath will be automatically connected to the `is_b_zero` port in the control unit.

The PyMTL model is in `GcdUnitCL.py` and the corresponding test script is in `GcdUnitCL_test.py`. As with the GCD unit CL model, our RTL model is able use the exact same test setup as the GCD unit

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

class GcdUnitDpathRTL (Model):

  # Constructor

  def __init__( 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  (1)
    s.b_mux_sel = InPort  (B_MUX_SEL_NBITS)
    s.b_reg_en  = InPort  (1)

    # Status signals (dpath -> ctrl)

    s.is_b_zero = OutPort (1)
    s.is_a_lt_b = OutPort (1)

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

    # A mux

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

    s.a_mux = m = Mux( 16, 3 )
    s.connect_pairs(
      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(16)
    s.connect_pairs(
      m.en,  s.a_reg_en,
      m.in_, s.a_mux.out,
    )
```

Figure 51: 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 `s.connect_pair` to enable more succinct structural composition in datapaths.

```
1  cycle src        A    B    Areg Breg ST out    sink
2  -------------------------------------------------
3     2:            >          (0005 000f I )     >
4     3: 000f0005 > 0005:000f(0005 000f I )     >
5     4: #          > #        (0005 000f Cs)     >
6     5: #          > #        (000f 0005 C-)     >
7     6: #          > #        (000a 0005 C-)     >
8     7: #          > #        (0005 0005 C-)     >
9     8: #          > #        (0000 0005 Cs)     >
10    9: #          > #        (0005 0000 C )     >
11   10: #          > #        (0005 0000 D )0005 > 0005
12   11: 00030009 > 0009:0003(0005 0000 I ).    > .
13   12: #          > #        (0009 0003 C-)     >
14   13: #          > #        (0006 0003 C-)     >
15   14: #          > #        (0003 0003 C-)     >
16   15: #          > #        (0000 0003 Cs)     >
17   16: #          > #        (0003 0000 C )     >
18   17: #          > #        (0003 0000 D )0003 > 0003
19   18: 00000000 > 0000:0000(0003 0000 I ).    > .
20   19: #          > #        (0000 0000 C ).    > .
21   20: #          > #        (0000 0000 D )#    > #
22   21: #          > #        (0000 0000 D )#    > #
23   22: #          > #        (0000 0000 D )#    > #
24   23: #          > #        (0000 0000 D )0000 > 0000
25   24: 001b000f > 000f:001b(0000 0000 I ).    > .
```

Figure 52: 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.

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 random delays in the test sink like this:

```
% cd ${TUTROOT}/build
% py.test ../tut3_pymtl/gcd/test/GcdUnitRTL_test.py -v
% py.test ../tut3_pymtl/gcd/test/GcdUnitRTL_test.py -sv -k basic_0x0
```

Figure 52 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 3. 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 17. Compare this to the line trace from our GCD unit CL model shown in Figure 48. 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 16 instead of cycle 17. 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.

★ *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 `py.test`:

```
% cd ${TUTROOT}/build
% py.test ../tut3_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
% ../tut3_pymtl/gcd/gcd-sim --stats --impl cl  --input random
% ../tut3_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 do not match exactly. You can generate the Verilog and waveforms to drive an FPGA or ASIC toolflow using the simulator like this:

```
% cd ${TUTROOT}/build
% ../tut3_pymtl/gcd/gcd-sim --impl rtl --input random --translate --dump-vcd
% ../tut3_pymtl/gcd/gcd-sim --impl rtl --input small  --translate --dump-vcd
% ../tut3_pymtl/gcd/gcd-sim --impl rtl --input zeros  --translate --dump-vcd
```

7. TravisCI for Continuous Integration

As discussed in the Git tutorial, TravisCI is an online continuous integration service that is tightly coupled to GitHub. TravisCI will automatically run all tests for a students' lab assignment every time the students push their code to GitHub. We will be using the results reported by TravisCI to

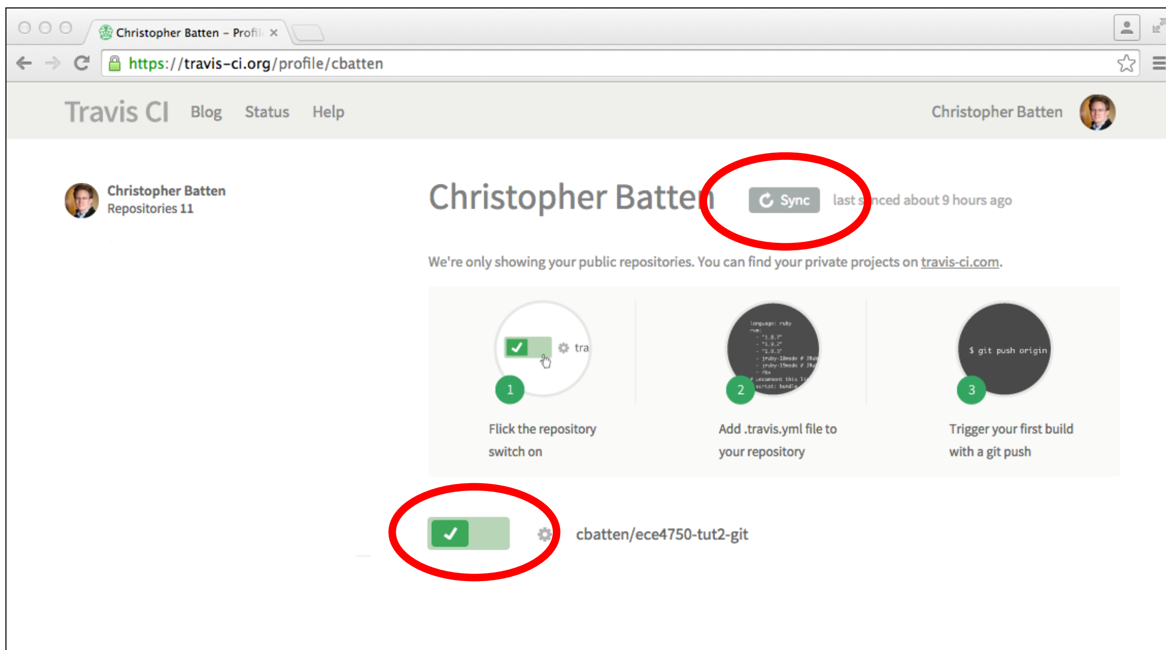

Figure 53: TravisCI Settings Page

evaluate the code functionality of the lab assignments. In this section, we do a small experiment to illustrate how TravisCI works for PyMTL projects.

The first step is to enable TravisCI for the remote repository in GitHub. Log into TravisCI using your GitHub ID and password:

- `https://travis-ci.org/profile`

Once you have signed in, you should go to your TravisCI profile and find the list of your public GitHub repositories. You may need to click *Sync* to ensure that TravisCI has the most recent view of your public repositories on GitHub. Turn on TravisCI with the little "switch" next to the repository we have been using in this tutorial (`<githubid>/ece4750-tut3-pymtl`). Figure 53 shows what the TravisCI settings page should look like and the corresponding "switch". After enabling TravisCI for the `<githubid>/ece4750-tut3-pymtl` repository, you should be able to go to the TravisCI page for this repository:

- `https://travis-ci.org/<githubid>/ece4750-tut3-pymtl`

TravisCI will report that there are no builds for this repository yet. Go ahead and commit all of the work you have done in this tutorial, then push your local commits to the remote repository on GitHub. If you revisit the TravisCI page for this repository, you should see TravisCI starting to build and run all of your tests. Figure 54 shows what the TravisCI build log will look like for a brand new fork of the tutorial repository. Study the TravisCI log output to verify that TravisCI is: (1) installing Verilator; (2) installing PyMTL; (3) creating a build directory; and (4) running all of your unit tests. Confirm that if all of the tests pass on `ecelinux` then they also pass on TravisCI.

```
Remove Log   Download Log

1   Using worker: worker-linux-docker-4291e1ab.prod.travis-ci.org:travis-linux-5
2
3   Build system information                                                          system_info
65
66  $ git clone --depth=50 --branch=master git://github.com/cbatten/ece4750-tut3-pymtl.git   git.checkout   0.39s
76
77  This job is running on container-based infrastructure, which does not allow use of 'sudo', setuid and setguid
    executables.
78  If you require sudo, add 'sudo: required' to your .travis.yml
79  See http://docs.travis-ci.com/user/workers/container-based-infrastructure/ for details.
80  $ source ~/virtualenv/python2.7/bin/activate                                                     0.00s
81  $ python --version
82  Python 2.7.9
83  $ pip --version
84  pip 6.0.7 from /home/travis/virtualenv/python2.7.9/lib/python2.7/site-packages (python 2.7)
85  $ wget --quiet http://brg.csl.cornell.edu/artifacts/verilator-travis-3.876.tar.gz   install.1   7.94s
86  $ tar xfz verilator-travis-3.876.tar.gz                                              install.2   1.46s
87  $ export VERILATOR_ROOT=${PWD}/verilator-3.876                                       install.3   0.00s
88  $ export PATH=${VERILATOR_ROOT}/bin:$PATH                                            install.4   0.00s
89  $ export PYMTL_VERILATOR_INCLUDE_DIR=${VERILATOR_ROOT}/include                       install.5   0.00s
90  $ verilator --version                                                                install.6   0.07s
92  $ pip -q install git+https://github.com/cornell-brg/pymtl.git@ece4750                install.7   6.85s
95  $ mkdir -p sim/build                                                                 before_script.1   0.00s
96  $ cd sim/build                                                                       before_script.2   0.00s
97  $ py.test ..                                                                                     7.67s
98  ============================= test session starts ==============================
99  platform linux2 -- Python 2.7.9 -- py-1.4.26 -- pytest-2.6.4
100 plugins: xdist, xdist, xdist
101 collected 95 items
102
103 ../tut3_pymtl/gcd/GcdUnitCL_test.py ......
104 ../tut3_pymtl/gcd/GcdUnitFL_test.py .....
105 ../tut3_pymtl/gcd/GcdUnitMsg_test.py ...
106 ../tut3_pymtl/gcd/GcdUnitRTL_test.py .....
107 ../tut3_pymtl/gcd/gcd_sim_test.py .........
108 ../tut3_pymtl/regincr/RegIncr2stage_test.py FFFF
109 ../tut3_pymtl/regincr/RegIncrNstage_test.py FFFFFFFFFFFFFF
110 ../tut3_pymtl/regincr/RegIncr_extra_test.py FF
111 ../tut3_pymtl/regincr/RegIncr_test.py .
112 ../tut3_pymtl/sort/SortUnitCL_test.py ...................
113 ../tut3_pymtl/sort/SortUnitFL_test.py ....
114 ../tut3_pymtl/sort/SortUnitFlatRTL_test.py .....
115 ../tut3_pymtl/sort/SortUnitFlatRTL_v_test.py .
116 ../tut3_pymtl/sort/SortUnitStructRTL_test.py FFFFF
117 ../tut3_pymtl/sort/sort_sim_test.py ............
118
119 ==================== 25 failed, 70 passed in 7.44 seconds ====================
120
121 The command "py.test .." exited with 1.
122
123 Done. Your build exited with 1.
                                                                                     Top
```

Figure 54: TravisCI Log

Acknowledgments

This tutorial was developed for ECE 4750 Computer Architecture course 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.

Appendix A: Constructs Allowed in Synthesizable Concurrent Blocks

<table>
<tr><th>Always Allowed in Synthesizable Concurrent Blocks</th><th>Allowed in Synthesizable Concurrent Blocks With Limitations</th><th>Explicitly Not Allowed in Synthesizable Concurrent Blocks</th></tr>
<tr><td><code>Bits</code></td><td>accessing Python lists[2]</td><td><code>* / // % **</code></td></tr>
<tr><td><code>BitStruct</code></td><td>writing signals</td><td><code>+= -= *= /= %= **= //=</code>[7]</td></tr>
<tr><td><code>& | ^ ^~ ~</code></td><td>with <code>.value/.next</code>[3]</td><td><code>for, while, break, continue</code></td></tr>
<tr><td><code>and or not</code></td><td>writing temporary variables[4]</td><td><code>def, global, class</code></td></tr>
<tr><td><code>+ -</code></td><td>reading <code>reset</code> signal[5]</td><td><code>try, except, raise</code></td></tr>
<tr><td><code>>> <<</code></td><td>read-modify-write <code>signal</code>[6]</td><td><code>as, is, in</code></td></tr>
<tr><td><code>== != > <= < <=</code></td><td></td><td><code>with, return, yield</code></td></tr>
<tr><td><code>reduce_and(), reduce_or()</code></td><td></td><td><code>import, from</code></td></tr>
<tr><td><code>reduce_xor()</code></td><td></td><td><code>del, exec, pass</code></td></tr>
<tr><td><code>sext(), zext(), concat()</code></td><td></td><td><code>lambda</code></td></tr>
<tr><td><code>if, else, elif</code></td><td></td><td><code>finally</code></td></tr>
<tr><td><code>s.signal[n], s.signal[n:m]</code></td><td></td><td>constructing Python lists</td></tr>
<tr><td>reading constant variables</td><td></td><td>constructing/using Python dicts</td></tr>
<tr><td>reading signals[1]</td><td></td><td>reading/writing non-signals[8]</td></tr>
<tr><td></td><td></td><td>writing signals</td></tr>
<tr><td></td><td></td><td>without <code>.value/.next</code>[9]</td></tr>
<tr><td></td><td></td><td>reading/writing <code>clk</code> signal</td></tr>
<tr><td></td><td></td><td>writing <code>reset</code> signal</td></tr>
</table>

1 Signals are instances of `InPort`, `OutPort`, `InValRdyBundle`, `OutValRdyBundle`, or `Wire`. Signals can only communicate bit-specific value types (e.g., `Bits`, `BitStruct`).

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 only be written using `.value` in `s.combinational` concurrent blocks. Signals must only be written using `.next` in `s.tick_rtl` concurrent blocks.

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 `s.tick_rtl` concurrent block. Reading the `reset` signal in a `s.combinational` 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 `s.combinational` concurrent block. This is a combinational loop and does not model valid hardware. Read-modify-write is allowed in an `s.tick_rtl` concurrent block using `.next`, although we urge students to consider separating the sequential and combinational logic. Students can use an `s.combinational` concurrent block to read the signal, perform some arithmetic on the corresponding value, and then write a temporary wire; and use an `s.tick_rtl` concurrent block to flop the temporary wire into the destination signal.

7 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 `s.tick_rtl`, 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.

8 Students cannot use non-signals (i.e., normal Python variables) to communicate between concurrent blocks. Students must use instances of `InPort`, `OutPort`, `InValRdyBundle`, `OutValRdyBundle`, or `Wire`.

9 Writing a signal without using `.value` or `.next` is not synthesizable and will likely result in strange simulator behavior.