ECE 4750 Computer Architecture

Tutorial 3: Verilog Hardware Description Language

School of Electrical and Computer Engineering
Cornell University

revision: 2024-08-31-11-23

Contents

1. Introduction

In the lab assignments for this course, we will be using Verilog for register-transfer-level (RTL) modeling and Python for functional-level modeling, verification, and simulator harnesses. This tutorial briefly reviews the basics of the Verilog hardware description language, but primarily focuses on how we can integrate Verilog RTL modeling into our PyMTL3 framework. The tutorial also includes some information about the coding conventions we will be using in the course. We will be using several open-source packages and tools: Icarus Verilog (`iverilog`) for experimenting with basic Verilog syntax; Verilator (`verilator`) for converting our more complex Verilog models into C++ source code; PyMTL3 for easily simulating and interacting with these compiled Verilog models; and the `pytest` framework for powerful test-driven Python development. We will be using `surfer` (an extension to VS Code) to view and analyze waveforms. As such, we will be pushing VS Code as our main IDE this year. All tools are installed and available on the `ecelinux` machines. This tutorial assumes that students have completed the Linux and Git tutorials.

Before you begin, make sure that you have **logged into the `ecelinux` servers** as described in the remote access tutorial. You will need to open a terminal and be ready to work at the Linux command line. You can do this using any of the methods described in the remote access tutorial: Windows PowerShell, Mac OS X Terminal, VS Code, MobaXterm, or Mac Terminal w/ XQuartz. That said, we *strongly* recommend the Windows and/or Mac + VSCode option. This is the setup for which you will be able to get most support from the course staff. To follow along with the tutorial, type the commands without the `%` character (for the `bash` prompt) or the `>>>` characters (for the `python` interpreter prompt). In addition to working through the commands in the tutorial, you should also try the more open-ended tasks marked with the ★ symbol.

Before you begin, make sure that you have **sourced the setup-ece4750.sh script**. Sourcing the setup script sets up the environment required for this tutorial.

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

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

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-verilog`

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

```
% source setup-ece4750.sh
% mkdir -p ${HOME}/ece4750
% cd ${HOME}/ece4750
% git clone git@github.com:githubid/ece4750-tut3-verilog.git tut3
% cd tut3/sim
% TUTROOT=${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. Functional-, Cycle-, and Register-Transfer-Level Modeling

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

2.1. Comparison of FL, CL, and RTL Modeling

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

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

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

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

In this course, we will focus on FL and RTL models, but it is important to keep in mind that many computer architects exclusively use CL modeling to productively explore high-level microarchitectural trade-offs before potentially working with chip designers to do RTL modeling.

2.2. Verilog Modeling: Synthesizable vs. Non-Synthesizable RTL

Verilog is a powerful language that was originally intended for building simulators of hardware as opposed to models that could automatically be transformed into hardware (e.g., synthesized to an FPGA or ASIC). Given this, it is very easy to write Verilog 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 assertions and line tracing. Non-synthesizable Verilog modeling is also critical when implementing Verilog test harnesses. **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 register-transfer-level (RTL) models. It is acceptable to include a limited amount of non-synthesizable code in students' designs for the sole purpose of debugging, assertions, or line tracing. If a student includes non-synthesizable code in the actual design, they must explicitly demarcate this code by wrapping it in `` `ifndef SYNTHESIS ``

and `` `endif ``. 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 Verilog constructs are allowed in synthesizable RTL, which constructs are allowed in synthesizable RTL with limitations, and which constructs are explicitly not allowed in synthesizable RTL. There are no limits on using the Verilog preprocessor, since the preprocessing step happens at compile time.

3. Verilog Basics: Data Types, Operators, and Conditionals

We will begin by writing some very basic code to explore Verilog data types, operators, and conditionals. We will not be modeling actual hardware yet; we are just experimenting with the language. Start by creating a build directory to work in.

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

3.1. Hello World

Create a new Verilog source file named `hello-world.v` with the contents shown in Figure 1 using your favorite text editor. A `module` is the fundamental hardware building block in Verilog, but for now we are simply using it to encapsulate an `initial` block. The `initial` block specifies code which should be executed "at the beginning of time" when the simulator starts. Since real hardware cannot do anything "at the beginning of time" `initial` blocks are not synthesizable and you should not use them in the synthesizable portion of your designs. However, `initial` blocks can be useful for test harnesses and when experimenting with the Verilog language. The `initial` block in Figure 1 contains a single call to the `display` system task which will output the given string to the console.

We will be using `iverilog` to compile Verilog models into simulators in the beginning of this tutorial before we turn our attention to using Verilator. You can run `iverilog` as follows to compile `hello-world.v`.

```
% cd ${TUTROOT}/build
% iverilog -g2012 -o hello-world hello-world.v
```

The `-g2012` option tells `iverilog` to accept newer SystemVerilog syntax, and the `-o` specifies the name of the simulator that `iverilog` will create. You can run this simulator as follows.

```
% cd ${TUTROOT}/build
% ./hello-world
```

```
1 module top;
2   initial begin
3     $display( "Hello World!" );
4   end
5 endmodule
```

Code at `https://github.com/cbatten/x/blob/master/ex-basics-hello-world.v`

Figure 1: Verilog Basics: Display Statement – The obligatory "Hello, World!" program to compiling a basic Verilog program.

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

Arithmetic Operators	
`+`	addition
`-`	subtraction

Reduction Operators	
`&`	reduce via AND
`~&`	reduce via NAND
`|`	reduce via OR
`~|`	reduce via NOR
`^`	reduce via XOR
`^~`	reduce via XNOR

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

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

Other Operators	
`{}`	concatenate
`{N{}}`	replicate N times

Table 1: Table of Verilog Operators – Not all Verilog operators are shown, just those operators that are acceptable for use in the synthesizable RTL portion of students' designs.

As discussed in the Linux tutorial you can compile the Verilog and run the simulator in a single step.

```
% cd ${TUTROOT}/build
% iverilog -g2012 -o hello-world hello-world.v && ./hello-world
```

This makes it easy to edit the Verilog source file, quickly recompile, and test your changes by switching to your terminal, pressing the up-arrow key, and then pressing enter.

★ *To-Do On Your Own:* Edit the string that is displayed in this simple program, recompile, and rerun the simulator.

3.2. Logic Data Types

To understand any new modeling language we usually start by exploring the primitive data types for representing values in a model. Verilog uses a combination of the `wire` and `reg` keywords which interact in subtle and confusing ways. SystemVerilog has simplified modeling by introducing the `logic` data type. We will be exclusively using `logic` as the general-purpose, hardware-centric data type for modeling a single bit or multiple bits. Each bit can take on one of four values: 0, 1, X, Z. X is used to represent unknown values (e.g., the state of a register on reset). Z is used to represent high-impedance values. Although we will use variables with X values in this course, we will not use variables with Z values (although you may see Z values if you forget to connect an input port of a module).

Table 1 shows the operators that we will be primarily using in this course. Note that Verilog and SystemVerilog support additional operators including `*` for multiplication, `/` for division, `%` for modulus, `**` for exponent, and `===`/`!===` for special equality checks. There may occasionally be reasons to use one of these operators in your assertion or line tracing logic. However, these operators are either not synthesizable or synthesize poorly, so students are not allowed to use these operators in the synthesizable portion of their designs.

```
module top;

  // Declare single-bit logic variables.

  logic a;
  logic b;
  logic c;

  initial begin

    // Single-bit literals

    a = 1'b0;    $display( "1'b0   = %x ", a );
    a = 1'b1;    $display( "1'b1   = %x ", a );
    a = 1'bx;    $display( "1'bx   = %x ", a );
    a = 1'bz;    $display( "1'bz   = %x ", a );

    // Bitwise logical operators for doing AND, OR, XOR, and NOT

    a = 1'b0;
    b = 1'b1;

    c = a & b;   $display( "0 & 1  = %x ", c );
    c = a | b;   $display( "0 | 1  = %x ", c );
    c = a ^ b;   $display( "0 ^ 1  = %x ", c );
    c = ~b;      $display( "~1     = %x ", c );

    // Bitwise logical operators for doing AND, OR, XOR, and NOT with X

    a = 1'b0;
    b = 1'bx;

    c = a & b;   $display( "0 & x  = %x ", c );
    c = a | b;   $display( "0 | x  = %x ", c );
    c = a ^ b;   $display( "0 ^ x  = %x ", c );
    c = ~b;      $display( "~x     = %x ", c );

    // Boolean logical operators

    a = 1'b0;
    b = 1'b1;

    c = a && b;  $display( "0 && 1 = %x ", c );
    c = a || b;  $display( "0 || 1 = %x ", c );
    c = !b;      $display( "!1     = %x ", c );

  end

endmodule
```

Code at `https://github.com/cbatten/x/blob/master/ex-basics-logic-sbit.v`

Figure 2: Verilog Basics: Single-Bit Logic and Logical Operators – Experimenting with single-bit logic variables and literals, logical bitwise operators, and logical boolean operators.

Figure 2 shows an example program that illustrates single-bit logic types. Create a new Verilog source file named `logic-sbit.v` and copy some or all of this code. Compile this source file and run the resulting simulator.

Lines 13–16 illustrate how to write single-bit literals to express constant values. Lines 23–26 illustrate basic bitwise logical operators (`&`, `|`, `^`, `~`). Whenever we consider an expression in Verilog, we should always ask ourselves, "What will happen if one of the inputs is an X?" Lines 33–36 illustrate what happens if the second operand is an X for bitwise logical operators. Recall that X means "unknown". If we OR the value 0 with an unknown value we cannot know the result. If the unknown value is 0, then the result should be 0, but if the unknown value is 1, then the result should be 1. So Verilog specifies that in this case the value of the expression is X. Notice what happens if we AND the value 0 with an unknown value. In this case, we can guarantee that for any value for the second operand the result will always be 0, so Verilog specifies the value of the expression is 0.

In addition to the basic bitwise logical operators, Verilog also defines three Boolean logical operators (`&&`, `||`, `!`). These operators are effectively the same as the basic logical operators (`&`, `|`, `~`) when operating on single-bit logic values. The difference is really in the designer's intent. Using `&&`, `||`, `!` suggests that the designer is implementing a Boolean logic expression as opposed to doing low-level bit manipulation.

★ *To-Do On Your Own:* Experiment with more complicated multi-stage logic expressions by writing the Boolean logic equations for a one-bit full-adder. Use the `display` system task to output the result to the console. Experiment with using X input values as inputs to these logic expressions.

Multi-bit logic types are used for modeling bit vectors. Figure 3 shows an example program that illustrates multi-bit logic types. Create a new Verilog source file named `logic-mbit.v` and copy some or all of this code. Compile this source file and run the resulting simulator.

Lines 5–8 declares multi-bit logic variables. The square brackets contain the index of the most-significant and the least-significant bit. In this course, you should always use zero as the index of the least significant bit. Note that to declare a four-bit logic value, we use `[3:0]` not `[4:0]`.

Lines 14–17 illustrate multi-bit literals that can be used to declare constant values. These literals have the following general syntax: `<bitwidth>'<base><number>` where `<base>` can be `b` for binary, `h` for hexadecimal, or `d` for decimal. It is legal to include underscores in the literal, which can be helpful for improving the readability of long literals.

Lines 24–28 illustrate multi-bit versions of the basic bitwise logic operators. As before, we should always ask ourselves, "What will happen if one of the inputs is an X?" Lines 35–39 illustrate what happens if two bits in the second value are Xs. Note that some bits in the result are X and some can be guaranteed to be either a 0 or 1.

Lines 45–50 illustrate the reduction operators. These operators take a multi-bit logic value and combine all of the bits into a single-bit value. For example, the OR reduction operator (`|`) will OR all of the bits together.

★ *To-Do On Your Own:* We will use relational operators (e.g., ==) to compare two multi-bit logic values, but see if you can achieve the same effect with the bitwise XOR/XNOR operators and OR/NOR reduction operators.

```
module top;

  // Declare multi-bit logic variables

  logic [ 3:0] A; // 4-bit  logic variable
  logic [ 3:0] B; // 4-bit  logic variable
  logic [ 3:0] C; // 4-bit  logic variable
  logic [11:0] D; // 12-bit logic variable

  initial begin

    // Multi-bit literals

    A = 4'b0101;             $display( "4'b0101            = %x", A );
    D = 12'b1100_1010_0101;  $display( "12'b1100_1010_0101 = %x", D );
    D = 12'hca5;             $display( "12'hca5            = %x", D );
    D = 12'd1058;            $display( "12'd1058           = %x", D );

    // Bitwise logical operators for doing AND, OR, XOR, and NOT

    A = 4'b0101;
    B = 4'b0011;

    C = A & B;    $display( "4'b0101 &  4'b0011 = %b", C );
    C = A | B;    $display( "4'b0101 |  4'b0011 = %b", C );
    C = A ^ B;    $display( "4'b0101 ^  4'b0011 = %b", C );
    C = A ^~ B;   $display( "4'b0101 ^~ 4'b0011 = %b", C );
    C = ~B;       $display( "~4'b0011           = %b", C );

    // Bitwise logical operators when some bits are X

    A = 4'b0101;
    B = 4'b00xx;

    C = A & B;    $display( "4'b0101 &  4'b00xx = %b", C );
    C = A | B;    $display( "4'b0101 |  4'b00xx = %b", C );
    C = A ^ B;    $display( "4'b0101 ^  4'b00xx = %b", C );
    C = A ^~ B;   $display( "4'b0101 ^~ 4'b00xx = %b", C );
    C = ~B;       $display( "~4'b00xx           = %b", C );

    // Reduction operators

    A = 4'b0101;

    C = &A;       $display( " & 4'b0101 = %b", C );
    C = ~&A;      $display( "~& 4'b0101 = %b", C );
    C = |A;       $display( " | 4'b0101 = %b", C );
    C = ~|A;      $display( "~| 4'b0101 = %b", C );
    C = ^A;       $display( "^  4'b0101 = %b", C );
    C = ~^A;      $display( "~^ 4'b0101 = %b", C );

  end

endmodule
```

Code at `https://github.com/cbatten/x/blob/master/ex-basics-logic-mbit.v`

Figure 3: Verilog Basics: Multi-Bit Logic and Logical Operators – Experimenting with multi-bit logic variables and literals, bitwise logical operators, and reduction operators.

3.3. Shift Operators

Figure 4 illustrates three shift operators on multi-bit logic values. Create a new Verilog source file named `logic-shift.v` and copy some or all of this code. Compile this source file and run the resulting simulator.

Notice how the logical shift operators (<<, >>) always shift in zeros, but the arithmetic right shift operator (>>>) replicates the most-significant bit. Verilog requires that the left-hand operand to the arithmetic shift operator be explicitly denoted as signed, which we have done using the `signed` system task. We recommend this approach and avoiding the use of signed data types.

```
1  module top;
2
3    logic [7:0] A;
4    logic [7:0] B;
5    logic [7:0] C;
6
7    initial begin
8
9      // Fixed shift amount for logical shifts
10
11     A = 8'b1110_0101;
12
13     C = A << 1;            $display( "8'b1110_0101 << 1 = %b", C );
14     C = A << 2;            $display( "8'b1110_0101 << 2 = %b", C );
15     C = A << 3;            $display( "8'b1110_0101 << 3 = %b", C );
16
17     C = A >> 1;            $display( "8'b1110_0101 >> 1 = %b", C );
18     C = A >> 2;            $display( "8'b1110_0101 >> 2 = %b", C );
19     C = A >> 3;            $display( "8'b1110_0101 >> 3 = %b", C );
20
21     // Fixed shift amount for arithmetic shifts
22
23     A = 8'b0110_0100;
24
25     C = $signed(A) >>> 1;  $display( "8'b0110_0100 >>> 1 = %b", C );
26     C = $signed(A) >>> 2;  $display( "8'b0110_0100 >>> 2 = %b", C );
27     C = $signed(A) >>> 3;  $display( "8'b0110_0100 >>> 3 = %b", C );
28
29     A = 8'b1110_0101;
30
31     C = $signed(A) >>> 1;  $display( "8'b1110_0101 >>> 1 = %b", C );
32     C = $signed(A) >>> 2;  $display( "8'b1110_0101 >>> 2 = %b", C );
33     C = $signed(A) >>> 3;  $display( "8'b1110_0101 >>> 3 = %b", C );
34
35     // Variable shift amount for logical shifts
36
37     A = 8'b1110_0101;
38     B = 8'd2;
39
40     C = A << B;            $display( "8'b1110_0101 << 2 = %b", C );
41     C = A >> B;            $display( "8'b1110_0101 >> 2 = %b", C );
42
43   end
44
45 endmodule
```

Code at `https://github.com/cbatten/x/blob/master/ex-basics-logic-shift.v`

Figure 4: Verilog Basics: Shift Operators – Experimenting with logical and arithmetic shift operators and fixed/variable shift amounts.

3.4. Arithmetic Operators

Figure 5 illustrates the addition and subtraction operators for multi-bit logic values. Create a new Verilog source file named `logic-arith.v` and copy some or all of this code. Compile this source file and run the resulting simulator.

These operators treat the multi-bit logic values as unsigned integers. Although Verilog does include support for signed arithmetic, these constructs may not be synthesizable so you are required to use only unsigned arithmetic. Also recall that `*`, `/`, `%`, `**` are not allowed in the synthesizable portion of your design.

Note that carefully considering the bitwidths of the input and output variables is important. Lines 22–23 illustrate overflow and underflow. You can see that if you overflow the bitwidth of the output variable then the result will simply wrap around. Similarly, since we are using unsigned arithmetic negative numbers wrap around. This is also called modular arithmetic.

★ *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, try `200 + 400 + 400 - 400 - 400`. Does this expression produce the expected answer even though the intermediate values overflowed?

```
1  module top;
2
3    logic [7:0] A;
4    logic [7:0] B;
5    logic [7:0] C;
6
7    initial begin
8
9      // Basic arithmetic with no overflow or underflow
10
11     A = 8'd28;
12     B = 8'd15;
13
14     C = A + B;  $display( "8'd28 + 8'd15 = %d", C );
15     C = A - B;  $display( "8'd28 - 8'd15 = %d", C );
16
17     // Basic arithmetic with overflow and underflow
18
19     A = 8'd250;
20     B = 8'd15;
21
22     C = A + B;  $display( "8'd250 + 8'd15  = %d", C );
23     C = B - A;  $display( "8'd15  - 8'd250 = %d", C );
24
25   end
26
27 endmodule
```

Code at `https://github.com/cbatten/x/blob/master/ex-basics-logic-arith.v`

Figure 5: Verilog Basics: Arithmetic Operators – Experimenting with arithmetic operators for addition and subtraction.

3.5. Relational Operators

Figure 6 illustrates the relational operators used for comparing two multi-bit logic values. Create a new Verilog source file named `logic-relop.v` and copy some or all of this code. Compile this source file and run the resulting simulator.

Lines 28–33 illustrate what happens if some of the bits are Xs for these relational operators. Notice that we can still determine two values are not equal even if some bits are unknown. If the bits we do know are different then the unknown bits do not matter; we can guarantee that the full bit vectors are not equal. So in this example, since we know that the top-two bits in `4'b1100` and `4'b10xx` then we can guarantee that the two values are not equal even though the bottom two bits of one operand are unknown.

The <, >, <=, >= operators behave slightly differently than the == and `!=` operators when considering values with Xs. In this example, we should be able to guarantee that `4'b1100` is always greater than `4'b10xx` (assuming these are unsigned values), since no matter what the bottom two bits are in the second operand it cannot be greater than the first operand. However, if you run this simulation, then you will see that the result is still X. This is not a bug and is correct according to the Verilog language specification. This is a great example of how Verilog has relatively complicated and sometimes inconsistent language semantics. Originally, there was no Verilog standard. The most common Verilog simulator was the de-factor language standard. I imagine the reason there is this difference between how == and < handle X values is simply because in the very first Verilog simulators it was the most efficient solution. These kind of "simulator implementation issues" can be found throughout the Verilog standard.

Lines 40–43 illustrates signed comparisons using the `signed` system task to interpret the unsigned input operands as signed values. To simplify our designs, we do not allow students to use signed types. We should explicitly use the `signed` system task whenever we need to ensure signed comparisons.

★ *To-Do On Your Own:* Try composing relational operators with the Boolean logic operators we learned about earlier in this section to create more complicated expressions.

```
1  module top;
2
3    // Declare multi-bit logic variables
4
5    logic        a; // 1-bit logic variable
6    logic [ 3:0] A; // 4-bit logic variable
7    logic [ 3:0] B; // 4-bit logic variable
8
9    initial begin
10
11     // Relational operators
12
13     A = 4'd15; B = 4'd09;
14
15     a = ( A == B );   $display( "(15 == 9) = %x", a );
16     a = ( A != B );   $display( "(15 != 9) = %x", a );
17     a = ( A >  B );   $display( "(15 >  9) = %x", a );
18     a = ( A >= B );   $display( "(15 >= 9) = %x", a );
19     a = ( A <  B );   $display( "(15 <  9) = %x", a );
20     a = ( A <= B );   $display( "(15 <= 9) = %x", a );
21
22     // Relational operators when some bits are X
23
24     A = 4'b1100; B = 4'b10xx;
25
26     a = ( A == B );   $display( "(4'b1100 == 4'b10xx) = %x", a );
27     a = ( A != B );   $display( "(4'b1100 != 4'b10xx) = %x", a );
28     a = ( A >  B );   $display( "(4'b1100 >  4'b10xx) = %x", a );
29     a = ( A <  B );   $display( "(4'b1100 <  4'b10xx) = %x", a );
30
31     // Signed relational operators
32
33     A = 4'b1111; // -1 in twos complement
34     B = 4'd0001; //  1 in twos complement
35
36     a = (         A  >         B  ); $display( "(-1 > 1) = %x", a );
37     a = (         A  <         B  ); $display( "(-1 < 1) = %x", a );
38     a = ( $signed(A) > $signed(B) ); $display( "(-1 > 1) = %x", a );
39     a = ( $signed(A) < $signed(B) ); $display( "(-1 < 1) = %x", a );
40
41   end
42
43 endmodule
```

Code at `https://github.com/cbatten/x/blob/master/ex-basics-logic-relop.v`

Figure 6: Verilog Basics: Relational Operators – Experimenting with relational operators.

3.6. Concatenation Operators

Figure 7 illustrates the concatenation operators used for creating larger bit vectors from multiple smaller bit vectors. Create a new Verilog source file named `logic-concat.v` and copy some or all of this code. Compile this source file and run the resulting simulator.

Lines 18–20 illustrate concatenating three four-bit logic variables and then assigning the result to a 12-bit logic variable. Lines 25–26 illustrate concatenating a four-bit logic variable with an eight-bit logic variable. The repeat operator can be used to duplicate a given logic variable multiple times when creating larger bit vectors. On Line 33, we replicate a four-bit logic value three times to create a 12-bit logic value.

```
1  module top;
2
3    logic [ 3:0] A; // 4-bit  logic variable
4    logic [ 3:0] B; // 4-bit  logic variable
5    logic [ 3:0] C; // 4-bit  logic variable
6    logic [ 7:0] D; // 8-bit  logic variable
7    logic [11:0] E; // 12-bit logic variable
8
9    initial begin
10
11     // Basic concatenation
12
13     A = 4'ha;
14     B = 4'hb;
15     C = 4'hc;
16     D = 8'hde;
17
18     E = { A, B, C };   $display( "{ 4'ha, 4'hb, 4'hc } = %x", E );
19     E = { C, A, B };   $display( "{ 4'hc, 4'ha, 4'hb } = %x", E );
20     E = { B, C, A };   $display( "{ 4'hb, 4'hc, 4'ha } = %x", E );
21
22     E = { A, D };      $display( "{ 4'ha,  8'hde } = %x", E );
23     E = { D, A };      $display( "{ 8'hde, 4'ha  } = %x", E );
24
25     E = { A, 8'hf0 };  $display( "{ 4'ha,  8'hf0 } = %x", E );
26     E = { 8'hf0, A };  $display( "{ 8'hf0, 4'ha  } = %x", E );
27
28     // Repeat operator
29
30     A = 4'ha;
31     B = 4'hb;
32
33     E = { 3{A} };      $display( "{ 4'ha, 4'ha, 4'ha } = %x", E );
34     E = { A, {2{B}} }; $display( "{ 4'ha, 4'hb, 4'hb } = %x", E );
35
36   end
37
38 endmodule
```

Code at `https://github.com/cbatten/x/blob/master/ex-basics-logic-concat.v`

Figure 7: Verilog Basics: Concatenation Operators – Experimenting with the basic concatenation operator and the repeat operator.

3.7. Enum Data Types

The `logic` data type will be the most common data type we use in our synthesizable RTL since a `logic` variable has a direct one-to-one correspondence with a bit vector in hardware. However, there are certain cases where using a `logic` variable can be quite tedious and error prone. SystemVerilog has introduced two new kinds of user-defined types that can greatly simplify some portions of our designs. In this subsection, we introduce the `enum` type which enables declaring variables that can only take on a predefined list of labels.

Figure 8 illustrates creating and using an `enum` type for holding a state variable which can take on one of four labels. Create a new Verilog source file named `enum.v` and copy all of this code. Compile this source file and run the resulting simulator.

Lines 3–8 declare a new `enum` type named `state_t`. Note that `state_t` is not a new *variable* but is instead a new *type*. We will use the `_t` suffix to distinguish type names from variable names. Note that after the `enum` keyword we have included a *base type* of `logic [$clog2(4)-1:0]`. This base type specifies how we wish variables of this new type to be stored. In this case, we are specifying that `state_t` variables should be encoded as a two-bit `logic` value. The `clog2` system task calculates the number of bits in the given argument; it is very useful when writing more parameterized code. So in this situation we just need to pass in the number of labels in the enum to `clog2`. SystemVerilog actually provides many different ways to create `enum` types including anonymous types, types where we do not specify the base type, or types where we explicitly define the value for each label. In this course, you should limit yourself to the exact syntax shown in this example.

Line 14 declares a new variable of type `state_t`. This is the first time we have seen a variable which has a type other than `logic`. The ability to introduce new user-defined types is one of the more powerful features of SystemVerilog. Lines 21–24 sets the `state` variable using the labels declared as part of the new `state_t` type. Lines 28–40 compare the value of the `state` variable with these same labels, and these comparisons can be used to take different action based on the current value.

There are several advantages to using an `enum` type compared to the basic `logic` type to represent a variable that can hold one of several labels including: (1) more directly capturing the designer's intent to improve code quality; (2) preventing mistakes by eliminating the possibility of defining labels with the same value or defining label values that are too large to fit in the underlying storage; and (3) preventing mistakes when assigning variables of a different type to an `enum` variable.

★ *To-Do On Your Own:* Create your own new `enum` type for the state variable we will use in the GCD example later in this tutorial. The new `enum` type should be called `state_t` and it should support three different labels: `STATE_IDLE`, `STATE_CALC`, `STATE_DONE`. Write some code to set and compare the value of a corresponding `state` variable.

```
// Declare enum type

typedef enum logic [$clog2(4)-1:0] {
  STATE_A,
  STATE_B,
  STATE_C,
  STATE_D
} state_t;

module top;

  // Declare variables

  state_t state;
  logic   result;

  initial begin

    // Enum lable literals

    state = STATE_A; $display( "STATE_A = %d", state );
    state = STATE_B; $display( "STATE_B = %d", state );
    state = STATE_C; $display( "STATE_C = %d", state );
    state = STATE_D; $display( "STATE_D = %d", state );

    // Comparisons

    state = STATE_A;

    result = ( state == STATE_A );
    $display( "( STATE_A == STATE_A ) = %x", result );

    result = ( state == STATE_B );
    $display( "( STATE_A == STATE_B ) = %x", result );

    result = ( state != STATE_A );
    $display( "( STATE_A != STATE_A ) = %x", result );

    result = ( state != STATE_B );
    $display( "( STATE_A != STATE_B ) = %x", result );

  end

endmodule
```

Code at `https://github.com/cbatten/x/blob/master/ex-basics-enum.v`

Figure 8: Verilog Basics: Enum Data Types – Experimenting with `enum` data types including setting the value of an `enum` using a label and using the equality operator.

3.8. Struct Data Types

User-defined structures are now supported in SystemVerilog. Figure 9 illustrates creating and using a `struct` type for holding a variable with predefined named bit fields. Create a new Verilog source file named `struct.v` and copy all of this code. Compile this source file and run the resulting simulator.

Lines 3–7 declare a new `struct` type named `point_t`. Again note that `point_t` is not a new *variable* but is instead a new `type`. As before we use the `_t` suffix to distinguish type names from variable names. Note that after the `struct` keyword we have included the `packed` keyword which specifies that variables of this type should have an equivalent underlying `logic` storage. SystemVerilog also includes support for unpacked structs, but in this course, you should limit yourself to the exact syntax shown in this example. In addition to declaring the name of the new `struct` type, we also declare the named bit fields within the new `struct` type. The order of these bit fields is important; the first field will go in the most significant position of the underlying `logic` storage, the second field will go in the next position, and so on.

Lines 13–14 declare two new variables of type `point_t`. Line 18 declares a new `logic` variable with a bitwidth large enough to hold a variable of type `point_t`. We can use the `bits` system task to easily determine the total number of bits required to store a `struct` type. Lines 24–26 set the three fields of the `point` variable and Lines 28–30 read these three fields in order to display them. Line 34 copies one `point` variable into another `point` variable. Line 42 and 49 illustrate how to convert a `point` variable to/from a basic `logic` variable.

There are several advantages to using a `struct` type compared to the basic `logic` type to represent a variable with a predefined set of named bit fields including: (1) more directly capturing the designer's intent to improve code quality; (2) reducing the syntactic overhead of managing bit fields; and (3) preventing mistakes in modifying bit fields and in accessing bit fields.

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

```
// Declare struct type

typedef struct packed {  // Packed format:
  logic [3:0] x;         //   11 8 7  4 3  0
  logic [3:0] y;         //  +----+----+----+
  logic [3:0] z;         //  | x  | y  | z  |
} point_t;               //  +----+----+----+

module top;

  // Declare variables

  point_t point_a;
  point_t point_b;

  // Declare other variables using $bits()

  logic [$bits(point_t)-1:0] point_bits;

  initial begin

    // Reading and writing fields

    point_a.x = 4'h3;
    point_a.y = 4'h4;
    point_a.z = 4'h5;

    $display( "point_a.x = %x", point_a.x );
    $display( "point_a.y = %x", point_a.y );
    $display( "point_a.z = %x", point_a.z );

    // Assign structs

    point_b = point_a;

    $display( "point_b.x = %x", point_b.x );
    $display( "point_b.y = %x", point_b.y );
    $display( "point_b.z = %x", point_b.z );

    // Assign structs to bit vector

    point_bits = point_a;

    $display( "point_bits = %x", point_bits );

    // Assign bit vector to struct

    point_bits = { 4'd13, 4'd9, 4'd3 };
    point_a = point_bits;

    $display( "point_a.x = %x", point_a.x );
    $display( "point_a.y = %x", point_a.y );
    $display( "point_a.z = %x", point_a.z );

  end

endmodule
```

Code at `https://github.com/cbatten/x/blob/master/ex-basics-struct.v`

Figure 9: Verilog Basics: Struct Data Types – Experimenting with `struct` data types including read/writing fields and converting to/from `logic` bit vectors.

3.9. Ternary Operator

Figure 10 illustrates using the `ternary` operator for conditional execution. Create a new Verilog source file named `ternary.v` and copy some or all of this code. Compile this source file and run the resulting simulator.

Lines 12–19 illustrate using the ternary operator to choose what value to assign to the logic variable `c`. We can nest multiple ternary operators to compactly create expressions with multiple conditions. Lines 23–31 illustrate using four levels of nesting to choose among four different values for assigning `c`.

Lines 35–53 illustrate how the ternary operator functions if the conditional is unknown. In lines 35–43, all bits of the conditional are unknown, while in lines 45–53 only one bit of the conditional is unknown. If you examine the output from this simulator, you will see that Verilog semantics require any bits which can be guaranteed to be either 0 or 1 to be set as such, while the remaining bits are set to X. Regardless of the condition, the upper five bits of `c` are guaranteed to be `00001`.

Note that the four ternary operators cover all possible combinations of the two-bit input, so the final value (i.e., `8'h0e`) will never be used. In other words, if the conditionals contain unknowns this does *not* mean the condition evaluates to false. This is very different from the `if` statements described in the next subsection.

Aside: For some reason, many students insist on writing code like this:

```
a = ( cond_a ) ? 1'b1 : 1'b0;
b = ( cond_b ) ? 1'b0 : 1'b1;
```

This obfuscates the code and is not necessary. We are using a ternary operator to simply choose between 0 or 1. You should just assign the result of the conditional expression to `a` and `b` like this:

```
a =  ( cond_a );
b = !( cond_b );
```

★ *To-Do On Your Own:* Experiment with different uses of the ternary operator.

```
1  module top;
2
3    logic [7:0] a;
4    logic [7:0] b;
5    logic [7:0] c;
6    logic [1:0] sel;
7
8    initial begin
9
10     // ternary statement
11
12     a = 8'd30;
13     b = 8'd16;
14
15     c = ( a < b ) ? 15 : 14;
16     $display( "c = %d", c );
17
18     c = ( b < a ) ? 15 : 14;
19     $display( "c = %d", c );
20
21     // nested ternary statement
22
23     sel = 2'b01;
24
25     c = ( sel == 2'b00 ) ? 8'h0a
26       : ( sel == 2'b01 ) ? 8'h0b
27       : ( sel == 2'b10 ) ? 8'h0c
28       : ( sel == 2'b11 ) ? 8'h0d
29       :                    8'h0e;
30
31     $display( "sel = 1, c = %b", c );
32
33     // nested ternary statement w/ X
34
35     sel = 2'bxx;
36
37     c = ( sel == 2'b00 ) ? 8'h0a
38       : ( sel == 2'b01 ) ? 8'h0b
39       : ( sel == 2'b10 ) ? 8'h0c
40       : ( sel == 2'b11 ) ? 8'h0d
41       :                    8'h0e;
42
43     $display( "sel = x, c = %b", c );
44
45     sel = 2'bx0;
46
47     c = ( sel == 2'b00 ) ? 8'h0a
48       : ( sel == 2'b01 ) ? 8'h0b
49       : ( sel == 2'b10 ) ? 8'h0c
50       : ( sel == 2'b11 ) ? 8'h0d
51       :                    8'h0e;
52
53     $display( "sel = x, c = %b", c );
54
55   end
56
57 endmodule
```

Code at `https://github.com/cbatten/x/blob/master/ex-basics-ternary.v`

Figure 10: Verilog Basics: Ternary Operator – Experimenting with the ternary operator including nested statements and what happens if the conditional includes an unknown.

3.10. If Statements

Figure 11 illustrates using `if` statements. Create a new Verilog source file named `if.v` and copy some or all of this code. Compile this source file and run the resulting simulator.

The `if` statement resembles similar constructs in many other programming languages. Lines 11–20 illustrate basic `if` statements and lines 24–33 illustrate `if/else` statements.

There are some subtle issues involved in how an `if` statement handles X values in the conditional. Lines 37–46 illustrate this issue. The `sel` value in this example is a single-bit X. What would we expect the value of `a` to be after this `if` statement? Since the conditional is unknown, we might expect any variables that are written from within the `if` statement to also be unknown. In other words, we might expect the value of `a` to be X after this `if` statement. If you run this example code, you will see that the value of `a` is actually `8'h0b`. This means that an X value in the conditional for an `if` statement is not treated as unknown but is instead simply treated as if the conditional evaluated to false! This issue is called *X optimism* since unknowns are essentially optimistically turned into known values. X optimism can cause subtle simulation/synthesis mismatch issues. If you are interested, there are several resources on the public course webpage that go into much more detail. For this course, we don't need to worry too much about X optimism since we are not actually synthesizing our designs.

★ *To-Do On Your Own:* Experiment with different `if` statements including nested `if` statements. Experiment with what happens when the conditional is unknown.

```
1  module top;
2
3    logic [7:0] a;
4    logic [7:0] b;
5    logic       sel;
6
7    initial begin
8
9      // if statement
10
11     a = 8'd30;
12     b = 8'd16;
13
14     if ( a == b ) begin
15       $display( "30 == 16" );
16     end
17
18     if ( a != b ) begin
19       $display( "30 != 16" );
20     end
21
22     // if else statement
23
24     sel = 1'b1;
25
26     if ( sel == 1'b0 ) begin
27       a = 8'h0a;
28     end
29     else begin
30       a = 8'h0b;
31     end
32
33     $display( "sel = 1, a = %x ", a );
34
35     // if else statement w/ X
36
37     sel = 1'bx;
38
39     if ( sel == 1'b0 ) begin
40       a = 8'h0a;
41     end
42     else begin
43       a = 8'h0b;
44     end
45
46     $display( "sel = x, a = %x ", a );
47
48   end
49
50 endmodule
```

Code at `https://github.com/cbatten/x/blob/master/ex-basics-if.v`

Figure 11: Verilog Basics: If Statements – Experimenting with `if` statements including what happens if the conditional includes an unknown.

3.11. Case Statements

Figure 12 illustrates using `case` statements. Create a new Verilog source file named `case.v` and copy some or all of this code. Compile this source file and run the resulting simulator.

The `case` statement resembles similar constructs in many other programming languages. Lines 12–22 illustrate a basic `case` statement where a two-bit `sel` variable is used to choose one of four case items.

There are similar issues as with the `if` statement in terms of how `case` statements handle X values in the conditional. In lines 26–36, the `sel` variable is set to all Xs. We might expect since the input to the `case` statement is unknown the output should also be unknown. However, if we look at the value of `a` after executing this case statement it will be `8'h0e`. In other words, if there is an X in the input to the `case` statement, then the case statement will fall through to the `default` case. In order to avoid *X optimism*, we recommend students always include a `default` case that sets all of the output variables to Xs.

Notice that it is valid syntax to use X values in the case items, as shown on lines 48–49. These will actually match Xs in the input condition, which is almost certainly not what you want. This does not model any kind of real hardware; we cannot check for Xs in hardware since in real hardware an unknown must be known (i.e., all Xs will either be a 0 or a 1 in real hardware). Given this, you should never use Xs in the case items for a `case` statement.

★ *To-Do On Your Own:* Experiment with a larger `case` statement for a `sel` variable with three instead of two bits.

```
1  module top;
2
3    // Declaring Variables
4
5    logic [1:0] sel;
6    logic [7:0] a;
7
8    initial begin
9
10     // case statement
11
12     sel = 2'b01;
13
14     case ( sel )
15       2'b00   : a = 8'h0a;
16       2'b01   : a = 8'h0b;
17       2'b10   : a = 8'h0c;
18       2'b11   : a = 8'h0d;
19       default : a = 8'h0e;
20     endcase
21
22     $display( "sel = 01, a = %x", a );
23
24     // case statement w/ X
25
26     sel = 2'bxx;
27
28     case ( sel )
29       2'b00   : a = 8'h0a;
30       2'b01   : a = 8'h0b;
31       2'b10   : a = 8'h0c;
32       2'b11   : a = 8'h0d;
33       default : a = 8'h0e;
34     endcase
35
36     $display( "sel = xx, a = %x", a );
37
38     // Do not use x's in the case
39     // selection items
40
41     sel = 2'bx0;
42
43     case ( sel )
44       2'b00   : a = 8'h0a;
45       2'b01   : a = 8'h0b;
46       2'b10   : a = 8'h0c;
47       2'b11   : a = 8'h0d;
48       2'bx0   : a = 8'h0e;
49       2'bxx   : a = 8'h0f;
50       default : a = 8'h00;
51     endcase
52
53     $display( "sel = x0, a = %x", a );
54
55   end
56
57 endmodule
```

Code at `https://github.com/cbatten/x/blob/master/ex-basics-case.v`

Figure 12: Verilog Basics: Case Statements – Experimenting with `case` statements including what happens if the selection expression and/or the case expressions includes an unknown.

3.12. Casez Statements

Figure 13 illustrates using `casez` statements. Create a new Verilog source file named `casez.v` and copy some or all of this code. Compile this source file and run the resulting simulator.

The `casez` statement is very different from what you might find in other programming languages. The `casez` statement is a powerful construct that can enable very concise hardware models, but must be used carefully. A `casez` statement enables a designer to do "wildcard" matching on the input variable. Lines 10–23 illustrate using a `casez` statement to implement a "leading-one detector". This kind of logic outputs the bit position of the least-significant one in the input variable. We can use `?` characters in the case items as wildcards that will match either a 0 or 1 in the input variable. So both `4'b0100` and `4'b1100` will match the fourth case item. Implementing similar functionality using a `case` statement would require 16 items. Besides being more verbose, using a `case` statement also opens up additional opportunities for errors.

A `casez` statement behaves similarly to a `case` statement when there are Xs in the input. Lines 27–40 illustrate a situation where two of the bits in the input variable are unknown. This will match the `default` case and the output will be Xs.

Aside: Verilog includes a `casex` statement which you should never use. The reasoning is rather subtle, but to be safe stick to using `casez` statement if you need wildcard matching (and *only* if you need wildcard matching).

★ *To-Do On Your Own:* Experiment with a larger `casez` statement to implement a leading-one detector for an input variable with eight instead of four bits. How many case items would we need if we used a `case` statement to implement the same functionality?

```
1  module top;
2
3    logic [3:0] a;
4    logic [7:0] b;
5
6    initial begin
7
8      // casez statement
9
10     a = 4'b0100;
11
12     casez ( a )
13
14       4'b0000 : b = 8'd0;
15       4'b???1 : b = 8'd1;
16       4'b??10 : b = 8'd2;
17       4'b?100 : b = 8'd3;
18       4'b1000 : b = 8'd4;
19
20       default : b = 8'hxx;
21     endcase
22
23     $display( "a = 4'b0100, b = %x", b );
24
25     // casez statement w/ Xs
26
27     a = 4'b01xx;
28
29     casez ( a )
30
31       4'b0000 : b = 8'd0;
32       4'b???1 : b = 8'd1;
33       4'b??10 : b = 8'd2;
34       4'b?100 : b = 8'd3;
35       4'b1000 : b = 8'd4;
36
37       default : b = 8'hxx;
38     endcase
39
40     $display( "a = 4'b01xx, b = %x", b );
41
42   end
43
44 endmodule
```

Code at `https://github.com/cbatten/x/blob/master/ex-basics-casez.v`

Figure 13: Verilog Basics: Casez Statements – Experimenting with `casez` statements to illustrate their use as priority selectors with wildcards.

4. Registered Incrementer

In this section, we will create our very first Verilog hardware model and learn how to test this module using ad-hoc testing, line tracing, waveforms, and a sophisticated Python-based unit testing framework. It is good design practice to usually draw some kind of diagram of the hardware we wish to model before starting to develop the corresponding Verilog model. This diagram 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 Verilog 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 14. In this section, you will be gradually adding code to what we provide you in the `tut3_verilog/regincr` subdirectory.

4.1. Modeling a Registered Incrementer

Figure 15 shows the Verilog code which corresponds to the diagram in Figure 14. Every Verilog file should begin with a header comment as shown on lines 1–9 in Figure 15. The header comment should identify the primary module in the file, and include a brief description of what the module does. Reserve discussion of the actual implementation for later in the file. In general, you should attempt to keep lines in your Verilog 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. Note that the code in Figure 15 is artificially narrow so we can display two code listings side-by-side. Lines 11–12 create an "include guard" using the Verilog pre-processor. An include guard ensures that even if we include this Verilog file multiple times the modules within the file will only be declared once. Without include guards, the Verilog compiler will likely complain that the same module has been declared multiple times. Make sure that you have the corresponding end of the include guard at the bottom of your Verilog source file as shown on line 43.

Unlike many modern programming languages, Verilog does not have a clean way to manage namespaces for macros and module names. This means that you use the same macro or module name in two different files it will create a namespace collision which can potentially be very difficult to debug. We will follow very specific naming conventions to eliminate any possibility of a namespace collision. Our convention will to use the subdirectory path as a prefix for all Verilog macro and module names. Since the registered incrementer is in the directory `tut3_verilog/regincr`, we will use `TUT3_VERILOG_REGINCR_` as a prefix for all macro names and `tut3_verilog_regincr_` as a prefix for all module names. You can see this prefix being used for the macros on lines 11–12 and for the module name on line 14. To reiterate, *Verilog macro and module name must use the subdirectory path as a prefix*. While a bit tedious, this is essential to avoiding namespace collisions. As an aside, SystemVerilog *does* include namespaces, but this feature is not supported by Verilator yet so we are not using it in the course.

We begin by identifying the module's interface which in this case will include an eight-bit input port, eight-bit output port, and a clock input. Lines 15–20 in Figure 15 illustrate how we represent this interface using Verilog. A common mistake is to forget the semicolon (`;`) on line 20. A couple of comments about the coding conventions that we will be using in this course. All module names should always include the subproject name as a prefix (e.g., `ex_regincr_`). The portion of the name after this prefix should usually use `CamelCaseNaming`; each word begins with a capital letter without any underscores (e.g., `RegIncr`). Port names (as well as variable and module instance names) should use `underscore_naming`; all lowercase with underscores to separate words. As shown on lines 16–19, ports should be listed one per line with a two space initial indentation. The bitwidth specifiers and port names should all line up vertically. As shown on lines 15 and 20, the opening and closing parenthesis should be on their own separate lines. Carefully group ports to help the reader understand

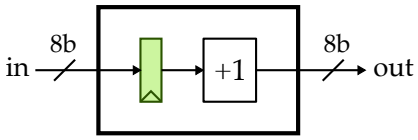

Figure 14: Block Diagram for Registered Incrementer – An eight-bit registered incrementer with an eight-bit input port, an eight-bit output port, and an (implicit) clock input.

```
1  //====================================
2  // Registered Incrementer
3  //====================================
4  // This is a simple example of a module
5  // for a registered incrementer which
6  // combines a positive edge triggered
7  // register with a combinational +1
8  // incrementer. We use flat register-
9  // transfer-level modeling.
10
11 `ifndef TUT3_VERILOG_REGINCR_REG_INCR_V
12 `define TUT3_VERILOG_REGINCR_REG_INCR_V
13
14 module tut3_verilog_regincr_RegIncr
15 (
16   input  logic       clk,
17   input  logic       reset,
18   input  logic [7:0] in_,
19   output logic [7:0] out
20 );
21
22   // Sequential logic
23
24   logic [7:0] reg_out;
25   always_ff @( posedge clk ) begin
26     if ( reset )
27       reg_out <= '0;
28     else
29       reg_out <= in_;
30   end
31
32   // Combinational logic
33
34   logic [7:0] temp_wire;
35   always_comb begin
36     temp_wire = reg_out + 1;
37   end
38
39   assign out = temp_wire;
40
41 endmodule
42
43 `endif /* TUT3_VERILOG_REGINCR_REG_INCR_V */
```

Figure 15: Registered Incrementer – An eight-bit registered +1 incrementer corresponding to the diagram in Figure 14.

```
1  //====================================
2  // Registered Incrementer
3  //====================================
4  // This is a simple example of a module
5  // for a registered incrementer which
6  // combines a positive edge triggered
7  // register with a combinational +1
8  // incrementer. We use flat register-
9  // transfer-level modeling.
10
11 `ifndef TUT3_VERILOG_REGINCR_REG_INCR_V
12 `define TUT3_VERILOG_REGINCR_REG_INCR_V
13
14 module tut3_verilog_regincr_RegIncr
15 (
16   input        clk,
17   input        reset,
18   input  [7:0] in_,
19   output [7:0] out
20 );
21
22   // Sequential logic
23
24   reg [7:0] reg_out;
25   always @( posedge clk ) begin
26     if ( reset )
27       reg_out <= 0;
28     else
29       reg_out <= in_;
30   end
31
32   // Combinational logic
33
34   reg [7:0] temp_wire;
35   always @(*) begin
36     temp_wire = reg_out + 1;
37   end
38
39   assign out = temp_wire;
40
41 endmodule
42
43 `endif /* TUT3_VERILOG_REGINCR_REG_INCR_V */
```

Figure 16: Registered Incrementer – An eight-bit registered +1 incrementer using Verilog-2001 constructs.

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`).

Lines 22–39 model the internal behavior of the module. We usually prefer using two spaces for each level of indentation; larger indentation can quickly result in significantly wasted horizontal space. *You should always use spaces and never insert any real tab characters into your source code.* You must limit yourself to synthesizable RTL for modeling your design. We will exclusively use two kinds of `always` blocks: `always_ff` concurrent blocks to model sequential logic and `always_comb` concurrent blocks to model combinational logic. We require students to clearly distinguishing between the portions of your code that are meant to model sequential logic from those portions meant to model combinational logic. This simple guideline can save significant frustration by making it easy to see subtle errors. For example, by convention we should only use non-blocking assignments in sequential logic (e.g., the <= operator on line 27) and we should only use blocking assignments in combinational logic (e.g., the = operator on line 36). We use the variable `reg_out` to hold the intermediate value between the register and the incrementer, and we use the variable `temp_wire` to hold the intermediate value between the incrementer and the output port. `reg_out` is modeling a register while `temp_wire` is modeling a wire. Notice that both of these variables use the `logic` data type; what makes one model a register while the other models a wire is how these variables are used. The sequential concurrent block update to `reg_out` means it models a register. The combinational concurrent block update to `temp_wire` means it models a wire.

The register incrementer illustrates the two fundamental ways to model combinational logic. We have used an `always_comb` concurrent block to model the actual incrementer logic and a continuous assignment statement (i.e., `assign`) to model connecting the temporary wire to the output port. We could just have easily written logic as part of the assign statement. For example, we could have used `assign out = reg_out + 1` and skipped the `always_comb` concurrent block. In general, we prefer continuous assignment statements over `always @(*)` concurrent blocks to model combinational logic, since it is easier to model less-realistic hardware using `always_comb` concurrent blocks. There is usually a more direct one-to-one mapping from continuous assignment statements to real hardware. However, there are many cases where it is significantly more convenient to use `always_comb` concurrent blocks or just not possible to use continuous assignment statements. Students will need to use their judgment to determine the most elegant way to represent the hardware they are modeling while still ensuring there is a clear mapping from the model to the target hardware.

A small aside about synchronous versus asynchronous resets. In general, we want to avoid reading the reset signal in an `always_comb` combinational block. 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. There may be some subtle cases where we need to factor the reset signal directly into combinational logic, but it should be rare.

Figure 15 illustrates a few new SystemVerilog constructs. Figure 16 illustrates the exact same registered incrementer implemented using the older Verilog-2001 hardware description language. Verilog-2001 uses `reg` and `wire` to specify variables instead of `logic`. All ports are of type `wire` by default. Determining when to use `reg` and `wire` is subtle and error prone. Note that `reg` is a misnomer; it does *not* model a register! On line 34, we must declare `temp_wire` to be of type `reg` even though it is modeling a wire. Verilog-2001 requires using `reg` for any variable written by an `always` concurrent block. Verilog-2001 uses a generic `always` block for both sequential and combinational concurrent blocks. While the `always @(*)` syntax is an improvement over the need in Verilog-1995 to explicitly define sensitivity lists, `always_ff` and `always_comb` more directly capture designer intent and allow

automated tools to catch common errors. For example, a Verilog simulator can catch errors where a designer accidentally uses a non-blocking assignment in an `always_comb` concurrent block, or where a designer accidentally writes the same `logic` variable from two different `always_comb` concurrent blocks. SystemVerilog is growing in popularity and increasingly becoming the de facto replacement for Verilog-2001, so it is worthwhile to carefully adopt new SystemVerilog features that can improve designer productivity.

Edit the Verilog source file named `RegIncr.v` in the `tut3_verilog/regincr` subdirectory using your favorite text editor. Add the combinational logic shown on lines 34–39 in Figure 16 which models the incrementer logic. We will be using `iverilog` to simulate this registered incrementer module, and `iverilog` does not currently support `always_ff` and `always_comb`, which is why we are using the Verilog-2001 construct for now.

4.2. Ad-Hoc Testing Using Verilog

Now that we have developed a new hardware model, our first thought should always turn to testing that model. Figure 17 shows an ad-hoc test for our registered incrementer using non-synthesizable Verilog. Note that we must explicitly include any Verilog files which contain modules that we want to use; Line 5 includes the Verilog source file that contains the registered incrementer. Lines 11–12 setup a clock with a period of 10 time steps. Notice that we are assigning an initial value to the `clk` net on line 11 and then modifying this net every five timesteps; setting initial values such as this is not synthesizable and should only be used in test harnesses. If you need to set an initial value in your design, you should use properly constructed reset logic.

Lines 20–26 instantiate the design under test. Notice that we use `underscore_naming` for the module instance name (e.g., `reg_incr`). You should almost always use named port binding (as opposed to positional port binding) to connect nets to the ports in a module instance. Lines 22–25 illustrate the correct coding convention with one port binding per line and the ports/nets vertically aligned. As shown on lines 21 and 26 the opening and closing parenthesis should be on their own separate lines. Although this may seem verbose, this coding style can significantly reduce errors by making it much easier to quickly visualize how ports are connected.

Lines 30–67 illustrate an `initial` block which executes at the very beginning of the simulation. `initial` blocks are not synthesizable and should only be used in test harnesses. Lines 34–35 instruct the simulator to dump waveforms for all nets. Line 39 is a delay statement that essentially waits for 11 timesteps. Delay statements are not synthesizable and should only be used in test harnesses. Lines 44–46 set the input, wait for 10 timesteps, and then displays the input and output port values.

Edit the Verilog simulation harness named `regincr-adhoc-test.v` in the `tut3_verilog/regincr` subdirectory using your favorite text editor. Add the code on lines 20–26 in Figure 17 to instantiate the registered incrementer model. Then use `iverilog` to compile this simulator and run the simulation as follows:

```
% cd ${TUTROOT}/tut3_verilog/regincr
% iverilog -g2012 -o regincr-adhoc-test regincr-adhoc-test.v
% ./regincr-adhoc-test
```

```
//========================================================================
// RegIncr Ad-Hoc Testing
//========================================================================

`include "../tut3_verilog/regincr/RegIncr.v"

module top;

  // Clocking

  logic clk = 1;
  always #5 clk = ~clk;

  // Instaniate the design under test

  logic       reset = 1;
  logic [7:0] in_;
  logic [7:0] out;

  tut3_verilog_regincr_RegIncr reg_incr
  (
    .clk   (clk),
    .reset (reset),
    .in_   (in_),
    .out   (out)
  );

  // Verify functionality

  initial begin

    // Dump waveforms

    $dumpfile("regincr-adhoc-test.vcd");
    $dumpvars;

    // Reset

    #11;
    reset = 1'b0;

    // Cycle 1

    in_ = 8'h01;
    #10;
    $display( " cycle = 1: in = %x, out = %x", in_, out );

    // Cycle 2

    in_ = 8'h13;
    #10;
    $display( " cycle = 2: in = %x, out = %x", in_, out );

    // Cycle 3

    in_ = 8'h25;
    #10;
    $display( " cycle = 3: in = %x, out = %x", in_, out );

    // Cycle 4

    in_ = 8'h37;
    #10;
    $display( " cycle = 4: in = %x, out = %x", in_, out );

    $finish;
  end

endmodule
```


Figure 17: Ad-Hoc Testing Using Verilog for Registered Incrementer – An ad-hoc test using Verilog for the eight-bit registered incrementer in Figure 16.

★ *To-Do On Your Own:* Edit the register incrementer so that it now increments by +2 instead of +1. Use an assign statement instead of the `always @(*)` concurrent block to do the incrementer logic. Recompile, rerun the ad-hoc test, and verify that the displayed output is as expected. When you are finished, edit the register incrementer so that it again increments by +1.

4.3. Ad-Hoc Testing Using Python

Writing test harnesses in Verilog is very tedious. There are some industry standard verification frameworks based on SystemVerilog, such as the Open Verification Methodology (OVM) and the Universal Verification Methodology (UVM), but these frameworks are very heavyweight and are not supported by open-source tools. In this course, we will be using Python to make sophisticated testing easy. More specifically, we will be using the PyMTL3 framework to write test harnesses (including using FL models as golden reference models) for our Verilog RTL models. PyMTL3 includes support for *Verilog import* by writing a special PyMTL3 wrapper model. Once we have created this wrapper model, we can use all of the power of Python for writing test harnesses.

Figure 18 illustrates such a PyMTL3 wrapper. On line 9, we import the Verilog backend related passes. On line 11, we inherit from both `VerilogPlaceholder` and `Component`. This is how we tell the `VerilogPlaceholderPass` that this is a special wrapper component. By default, the placeholder pass assumes the Verilog file to import is the same as the PyMTL3 component class name. For example, this `RegIncr` will import from `RegIncr.v`. In this tutorial, the Verilog models all have the prefix `tut3_verilog_<folder>_`. The placeholder pass will browse `sim/pymtl.ini` to see if it has `auto_prefix = yes`. If so, it will automatically append the folder prefix to the PyMTL3 component name so that the Verilog model can be properly imported. Here, the placeholder pass will use `tut3_verilog_regincr_RegIncr` as the Verilog model name. The PyMTL3 interface on lines 19–20 specifies all of the input and output ports for this component. We can use the `s.set_metadata` API to configure specific variables for the placeholder pass, the import pass, and the translation pass. In this example, we don't need any metadata to import the RegIncr model. The comments shows how to set the the port map and whether the Verilog model has clk or reset ports. It can be useful when you work on your own Verilog designs. Finally, we will see later in this tutorial how we can use line tracing within our Verilog modules. We will provide you an appropriate PyMTL3 wrapper for almost all of the components in this course so you can focus on RTL design and testing.

Figure 19 shows an ad-hoc test for our registered incrementer using Python which is similar to the ad-hoc test we saw in Figure 17. The Python script elaborates the registered incrementer model, creates a simulator, writes input values to the input ports, and displays the input/output ports. Line 13 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 17 adds three zero values to the end of the list so that our simulation will run for a few extra cycles before stopping. Lines 21–22 construct and elaborate the new `RegIncr` model. Lines 26–27 uses the `VerilogPlaceholderPass`, `VerilogTranslationImportPass`, and the `DefaultPassGroup` to add import the Verilog RTL model and add simulation facilities to the top-level component. We reset the simulator on line 32 which will raise the implicit reset signal for two cycles. Lines 36–49 define a loop that is used to iterate through the list of input values. For each input value, we write the value to the model's input port, display the values on the input/output ports, and tick the simulator. Note that we must use `@=` attribute when writing ports in the test script.

Edit the ad-hoc test named `regincr-adhoc-test.py`. Add the code on lines 21–22 in Figure 19 to construct and elaborate the model. Then run the simulator script as follows:

```
#=========================================================================
# 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 pymtl3 import *
from pymtl3.passes.backends.verilog import *

class RegIncr( VerilogPlaceholder, Component ):

  # Constructor

  def construct( s ):

    # Port-based interface

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

    # The port map by default uses the PyMTL3 port names
    # s.set_metadata( VerilogPlaceholderPass.port_map, {
    #   s.in_: 'in_',
    #   s.out: 'out',
    # })

    # has_clk and has_reset are True by default
    # s.set_metadata( VerilogPlaceholderPass.has_clk, True )
    # s.set_metadata( VerilogPlaceholderPass.has_reset, True )
```

Figure 18: Registered Incrementer Wrapper – PyMTL wrapper for the Verilog module shown in Figure 14.

```
% cd ${TUTROOT}/build
% python ../tut3_verilog/regincr/regincr-adhoc-test.py 0x01 0x13 0x25 0x37
```

Note that we are now working within a separate build directory. The process of compiling and running tests 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.

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

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

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

from sys     import argv
from RegIncr import RegIncr

# Get list of input values from command line

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

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

input_values.extend( [0]*3 )

# Instantiate and elaborate the model

model = RegIncr()
model.elaborate()

# Apply the Verilog import passes and the default pass group

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

# Reset simulator

model.sim_reset()

# Apply input values and display output values

for input_value in input_values:

  # Write input value to input port

  model.in_ @= input_value
  model.sim_eval_combinational()

  # Print input and output ports

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

  # Tick simulator one cycle

  model.sim_tick()
```

Figure 19: Ad-Hoc Testing Using Python for Registered Incrementer – Python script to elaborate the model, apply PyMTL3 passes, write input values to the input ports, and display the input/output ports.

★ *To-Do On Your Own:* Try running the ad-hoc test 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.

4.4. Visualizing a Model with Line Traces

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

To use line tracing, we need to add non-synthesizable Verilog to our RTL model that uses the Verilog macros and tasks provided in `vc/trace.v`. Figure 20 illustrates the registered incrementer now including basic line tracing code. First, we must include `vc/trace.v` on line 11. The actual line tracing code is on lines 44–54. This code must start with the `` `VC_TRACE_BEGIN `` macro (line 45) and end with the `` `VC_TRACE_END `` macro (line 50). Within the line tracing code, you can use `vc_trace.append_str(str)` to append a string to the current cycle's line trace. On line 47, we use the standard `$sformat` system task to create a string that includes the output the value of the input port, the value of the register output, and the value of the output port. On line 48, we append this string to the current cycle's line trace. Notice how the line tracing code uses `` `ifndef SYNTHESIS `` (line 42) and `` `endif `` (line 52) to indicate that this is non-synthesizable Verilog.

Go ahead and uncomment this line tracing code in `RegIncr.v`. To actually display the line trace, we also need to modify the `regincr-adhoc-test.py` script shown in Figure 19. First, remove the `print` statement on line 45. Then add `linetrace=True` as a keyword argument to the `DefaultPassGroup` on line 28 to enable the simulator to automatically display the line trace method.

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

Make these modifications and rerun the ad-hoc test. You should see the value at the input port, the current state of the register in the model, and the value at the output port:

```
1r 00 (00) 01
2r 00 (00) 01
3: 01 (00) 01
4: 13 (01) 02
5: 25 (13) 14
6: 37 (25) 26
7: 00 (37) 38
8: 00 (00) 01
9: 00 (00) 01
```

★ *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:

```
1r in:00 (00) out:01
2r in:00 (00) out:01
3: in:01 (00) out:01
4: in:13 (01) out:02
```

```
//========================================================================
// Registered Incrementer
//========================================================================
// This is a simple example of a module for a registered incrementer
// which combines a positive edge triggered register with a combinational
// +1 incrementer. We use flat register-transfer-level modeling.

`ifndef TUT3_VERILOG_REGINCR_REG_INCR_V
`define TUT3_VERILOG_REGINCR_REG_INCR_V

`include "vc/trace.v"

module tut3_verilog_regincr_RegIncr
(
  input        clk,
  input        reset,
  input  [7:0] in_,
  output [7:0] out
);

  // Sequential logic

  reg [7:0] reg_out;
  always @( posedge clk ) begin
    if ( reset )
      reg_out <= 0;
    else
      reg_out <= in_;
  end

  // Combinational logic

  reg [7:0] temp_wire;
  always @(*) begin
    temp_wire = reg_out + 1;
  end

  assign out = temp_wire;

  //------------------------------------------------------------------------
  // Line Tracing
  //------------------------------------------------------------------------

  `ifndef SYNTHESIS

  logic [`VC_TRACE_NBITS-1:0] str;
  `VC_TRACE_BEGIN
  begin
    $sformat( str, "%x (%x) %x", in_, reg_out, out );
    vc_trace.append_str( trace_str, str );
  end
  `VC_TRACE_END

  `endif /* SYNTHESIS */

endmodule

`endif /* TUT3_VERILOG_REGINCR_REG_INCR_V */
```

Figure 20: Registered Incrementer with Line Tracing – Line tracing displays trace output with each line corresponding to one (and only one!) cycle.

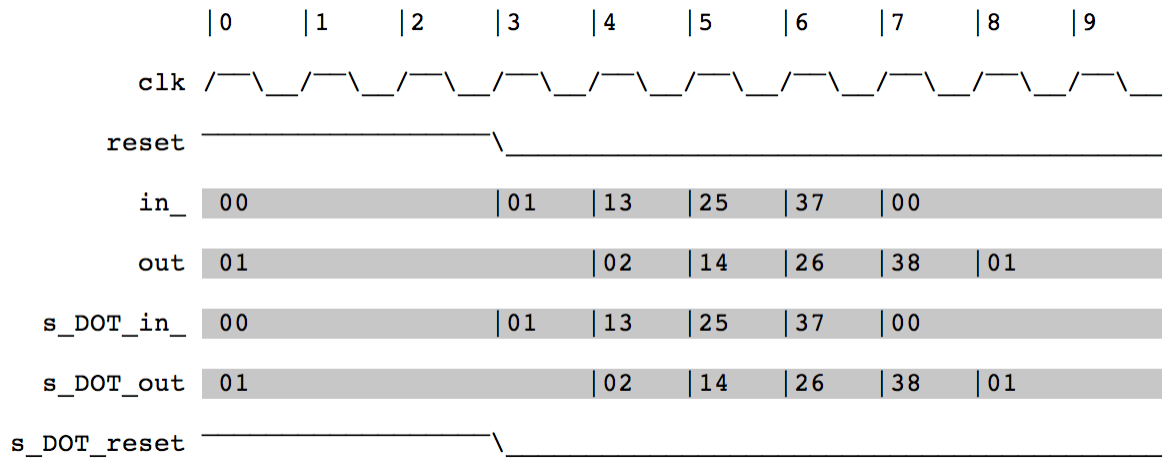

Figure 21: Text-Based Waveforms – Text-based waveforms are being used to display signals directly in the terminal associated with the registered incrementer shown in Figure 16.

4.5. Visualizing a Model with Text-Based Waveforms

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

To display the line trace, we need to modify the `regincr-adhoc-test.py` script shown in Figure 19. First, add `model.print_textwave()` after the loop at the very end of the script. This is because PyMTL3 doesn't want to dump the text-based waveform when your simulation is still going, so you need to call the `print_textwave` method explicitly when the simulation is finished. Then add `textwave=True` as a keyword argument to the `DefaultPassGroup` on line 28 to enable the simulator to automatically display the line trace method.

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

Make these modifications and rerun the ad-hoc test. Figure 21 shows screenshot of the terminal displaying the text-based waveform.

4.6. Visualizing a Model with VCD Waveforms

Line tracing can be useful for initially visualizing the high-level behavior of your design, and text-based waveforms is useful for visualizing very simple designs over just a few cycles. However, we often need to visualize many more signals than can be easily captured either in a line trace or text-based waveform. The PyMTL3 framework can output waveforms in the Value Change Dump (VCD) format for every signal (i.e., ports and wires) in your design which can then be visualized using a dedicated waveform viewer.

To generate VCD, we need to modify the `regincr-adhoc-test.py` script shown in Figure 19. Add `vcdwave="regincr-adhoc-test"` as a keyword argument to the `DefaultPassGroup` on line 28 to enable the simulator to write the VCD to a file named `regincr-adhoc-test.vcd`.

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

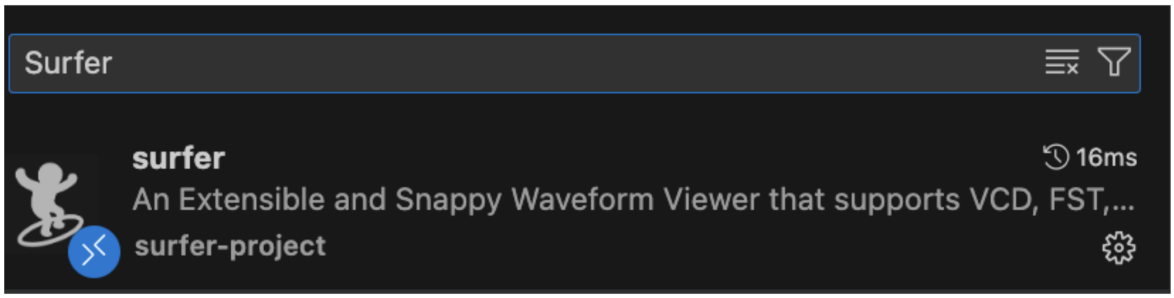

Figure 22: Surfer Waveform Viewer – Go to the extensions tab in VS Code, on the left panel, and search Surfer. Should be the first one. Click install on ssh .... as easy as that!

Figure 23: Surfer Waveform Viewer – Surfer is being used to browse the signals associated with the registered incrementer shown in Figure 16 and the ad-hoc test shown in Figure 24.

Make these modifications and rerun the ad-hoc test. Use Surfer to browse the generated waveforms as follows:

```
% cd ${TUTROOT}/build
% python ../tut3_verilog/regincr/regincr-adhoc-test.py 0x01 0x13 0x25 0x37
% code regincr-adhoc-test.vcd
```

You can browse the module hierarchy of your design in the left panel, with the signals in any given module being displayed in the lower-left panel. Add signs simply by clicking on them. To remove a signal, right click the signal in the wave viewer and click delete. You can use the scrollbar at the bottom to scroll to the right through the waveform or scroll in the viewer. You can use the view tab for zooming in/out or the corresponding magnifying glass icons in the toolbar to zoom in or out. Choose *File* > *Reload* (or click the circular arrow icon in the toolbar or press '`r`') to update the wave viewer after you have rerun a simulation. Figure 23 illustrates using Surfer to view the waveforms from our ad-hoc test. Surfer has many useful options which can make debugging your design more productive, so feel free to explore the associated documentation.

★ *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.

4.7. Verifying a Model with Unit Testing

Students might be tempted to use the ad-hoc testing approach illustrated in the previous sections as the primary way of testing their RTL designs. In other words, simply look at debug output, line traces, and/or waveforms 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 debug output 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 debug output and/or waveforms. Ad-hoc testing is usually verbose, which makes it error prone and more cumbersome to write tests. Ad-hoc testing is difficult for others to read and understand since by definition it is ad-hoc. Ad-hoc testing does not use any kind of standard test output, and does not provide support for controlling the amount of test output. While using ad-hoc testing 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.

In this course, we will be using the powerful `pytest` unit testing framework. The `pytest` framework is popular in the Python programming community with many features that make it well-suited for test-driven hardware development including: no-boilerplate testing with the standard `assert` statement; automatic test discovery; helpful traceback and failing assertion reporting; standard output capture; sophisticated parameterized testing; test marking for skipping certain tests; distributed testing; and many third-party plugins. More information is available at `http://www.pytest.org`.

Figure 24 illustrates a simple unit testing Python script for our registered incrementer. Notice at a high-level the test code is very straight-forward; the `pytest` framework enables unit testing to be as simple or as complex as necessary. The `pytest` framework includes automatic test discovery, which means that it will look through the unit test script and assume that any function that begins with `test_` is a test case. In this example, `pytest` will discover a single test case named `test_basic` corresponding to the function declared on lines 16–59. To test our registered incrementer, we need to instantiate and elaborate the model, use the default pass group to add simulation facilities, write values to the input ports of the model, and finally verify that the values read from the output ports of the model are correct.

Lines 20–24 instantiate and configure the model using the command line options. Lines 33–50 define a simple helper function that is responsible for verifying one cycle of execution. The helper function takes the desired test input and the reference test output as arguments. Line 37 writes the test input to the `in_` port of the registered incrementer. Note that it is important to use `@=` operator to write ports in the test harness. Line 41 tells the simulator to make sure any combinational blocks are executed if their input values have changed. Lines 45–46 read the `out` port and compare it to the reference output to ensure that the registered incrementer is functioning correctly. Notice that we check to make sure the reference output is not set to a question mark character. This gives us a simple way to indicate that we do not care what the output value is on that cycle. Also notice that the `pytest` framework does not need special assertion checking functions, and instead hooks into the standard `assert` statement provided in Python. This means the `pytest` framework can carefully track the `assert` statement on line 46, and on an assertion error will display the context of the `assert` statement including the sequence of function calls that lead to the assertion and the values of the variables used in the `assert` statement.

Lines 54–59 use our helper function to test the registered incrementer over six cycles. These test cases are an example of *directed cycle-by-cycle gray-box testing*. It is directed since we are explicitly creating directed tests as opposed to using some kind of random testing. It is cycle-by-cycle since we

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

from pymtl3 import *
from pymtl3.stdlib.test_utils import config_model_with_cmdline_opts

from ..RegIncr import RegIncr

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

def test_basic( cmdline_opts ):

  # Create the model

  model = RegIncr()

  # Configure the model

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

  # Create and reset simulator

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

  # Helper function

  def t( in_, out ):

    # Write input value to input port

    model.in_ @= in_

    # Ensure that all combinational concurrent blocks are called

    sim.sim_eval_combinational()

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

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

    # Tick simulator one cycle

    sim.sim_cycle()

  # Cycle-by-cycle tests

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

Figure 24: Unit Test Script for Registered Incrementer – A unit test for the eight-bit registered incrementer in Figure 16, which uses the `pytest` unit testing framework.

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

../tut3_pymtl/regincr/test/RegIncr_test.py .

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

Figure 25: `pytest` **Output –** Each line corresponds to one test script, and each dot corresponds to one passing test case. Failing test cases are shown with an `F` character.

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 24 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_simple_test.py`. Note that it is important that all test script file names end in `_test.py`, since this suffix is used by the `pytest` framework for automatic test discovery. Add the tests cases shown on lines 54–59 in Figure 24. We can run the test script using `pytest` as follows:

```
% cd ${TUTROOT}/build
% pytest ../tut3_verilog/regincr/test/RegIncr_simple_test.py
```

Again, notice that we run our unit test scripts from within a separate build directory to keep all generated files separate from the source tree. The `pytest` framework automatically discovers the `test_basic` test case. The output from running `pytest` should look similar to what is shown in Figure 25; `pytest` will display the name of the test script and a single dot indicating that the corresponding test case has passed. If we ran multiple test scripts, then each test script would have a separate line in the output. If we had multiple `test_` functions in `RegIncr_simple_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 25. This is because by default, the `pytest` framework "captures" the standard output from a test script instead of displaying this output. The output is only displayed when a test case fails, or if the users explicitly disables capturing the standard output. So to generate a line trace for this test, we simply use the `--capture=no` (or `-s`) command line option as follows:

```
% cd ${TUTROOT}/build
% pytest ../tut3_verilog/regincr/test/RegIncr_simple_test.py -s
```

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

```
% cd ${TUTROOT}/build
% pytest ../tut3_verilog/regincr/test/RegIncr_simple_test.py --tb=short
```

```
% pytest ../tut3_verilog/regincr/test/RegIncr_simple_test.py --tb=long
```

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

```
% cd ${TUTROOT}/build
% pytest ../tut3_verilog/regincr/test/RegIncr_simple_test.py --dump-vcd
% open tut3_pymtl.regincr.test.RegIncr_test__test_basic_top.verilator1.vcd &
```

Note that PyMTL3 will actually create *two* different VCD files. You always want to open the one with the `verilator1.vcd` suffix.

★ *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.

4.8. Verifying a Model with Test Vectors

The unit test shown in Figure 24 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 24, then each test case would need to duplicate lines 16–50. We could refactor this code into a separate helper function that can be reused across all test cases in a given test script. However, since this kind of testing is so common, PyMTL3 includes a flexible helper function for unit testing any model using test vectors. This function is named `run_test_vector_sim` and it is part of PyMTL3 Standard Library (`pymtl3.stdlib`).

Test vectors are essentially a table of test inputs and reference outputs. Figure 26 shows an extra test script that uses the `run_test_vector_sim` helper function provided by the PyMTL3 framework. There are three test cases for testing small input values, large input values, and the registered incrementer's overflow condition. The `run_test_vector_sim` helper function takes three arguments: an instantiated model, a test vector table, and the command line options. The function elaborates a model, uses the simulation tool to create a simulator, resets the simulator, writes the input values provided in the test vector table to the model's input ports, reads the values from the model's output ports, and compares the values to the reference values provided by the test vector table. The test vector table is a list of lists and is written so as to look like a table. Each column corresponds to either an input value or a reference output value, and each row corresponds to one cycle of the simulation. Question marks are allowed for reference output values when we don't care what the output is on that cycle. The first row of the test vector table is always a special "header string" that specifies the name of the model's input/output port for that column. Output ports are denoted with an asterisk suffix. Note how compact this test script is compared to the test script in Figure 24. This sophisticated helper function demonstrates the power of using a general-purpose dynamic language such as Python to write test harnesses.

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

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

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

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

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

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

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

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

Figure 26: Unit Test Script using Test Vectors for Registered Incrementer – A unit test for the eight-bit registered incrementer in Figure 16, which uses test vectors and the `pytest` unit testing framework.

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

../tut3_verilog/regincr/test/RegIncr2stage_test.py::test_small FAILED
../tut3_verilog/regincr/test/RegIncr2stage_test.py::test_large FAILED
../tut3_verilog/regincr/test/RegIncr2stage_test.py::test_overflow FAILED
../tut3_verilog/regincr/test/RegIncr2stage_test.py::test_random FAILED
../tut3_verilog/regincr/test/RegIncrNstage_test.py::test[2stage_small] FAILED
../tut3_verilog/regincr/test/RegIncrNstage_test.py::test[2stage_large] FAILED
../tut3_verilog/regincr/test/RegIncrNstage_test.py::test[2stage_overflow] FAILED
../tut3_verilog/regincr/test/RegIncrNstage_test.py::test[2stage_random] FAILED
../tut3_verilog/regincr/test/RegIncrNstage_test.py::test[3stage_small] FAILED
../tut3_verilog/regincr/test/RegIncrNstage_test.py::test[3stage_large] FAILED
../tut3_verilog/regincr/test/RegIncrNstage_test.py::test[3stage_overflow] FAILED
../tut3_verilog/regincr/test/RegIncrNstage_test.py::test[3stage_random] FAILED
../tut3_verilog/regincr/test/RegIncrNstage_test.py::test_random[1] PASSED
../tut3_verilog/regincr/test/RegIncrNstage_test.py::test_random[2] FAILED
../tut3_verilog/regincr/test/RegIncrNstage_test.py::test_random[3] FAILED
../tut3_verilog/regincr/test/RegIncrNstage_test.py::test_random[4] FAILED
../tut3_verilog/regincr/test/RegIncrNstage_test.py::test_random[5] FAILED
../tut3_verilog/regincr/test/RegIncrNstage_test.py::test_random[6] FAILED
../tut3_verilog/regincr/test/RegIncr_simple_test.py::test_basic PASSED
../tut3_verilog/regincr/test/RegIncr_test.py::test_small PASSED
../tut3_verilog/regincr/test/RegIncr_test.py::test_large PASSED

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

Figure 27: `pytest` **Verbose Output –** Each line corresponds to one test case. Passing test cases are marked with `PASSED` and failing test cases are marked with `FAILED`.

Edit the test script named `RegIncr_test.py`. Add the code on lines 35–46 in Figure 26 which tests for overflow. Run this test script using `pytest` as follows:

```
% cd ${TUTROOT}/build
% pytest ../tut3_verilog/regincr/test/RegIncr_test.py
```

The output should show the name of the test script and three dots corresponding to the three test cases in Figure 26. The `pytest` framework can automatically discover test scripts in addition to automatically discovering the test cases within a test script. If the argument to `pytest` is a directory, then `pytest` will search that directory for any files ending in `_test.py` and assume that these files are test scripts. The `pytest` framework also provides a more verbose output where each test case is listed on a separate line; passing test cases are marked with `PASSED` and failing test cases are marked with `FAILED`. Run both of the test scripts using the `--verbose` (or `-v`) command line option as follows:

```
% cd ${TUTROOT}/build
% pytest ../tut3_verilog/regincr/test -v
```

The verbose output should look similar to what is shown in Figure 27. Some test cases are passing for those models which we have completed, while other test cases are failing because we will work on them later in the tutorial. We can use the `-k` command line option to select just a few test cases to run and debug in more detail. For example to run just the test case for testing small input values, we can use the following:

```
% cd ${TUTROOT}/build
% pytest ../tut3_verilog/regincr/test -k small
```

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

```
% cd ${TUTROOT}/build
% pytest ../tut3_verilog/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
% pytest ../tut3_verilog/regincr/test/
% pytest ../tut3_verilog/regincr/test/RegIncr2stage_test.py -v
% pytest ../tut3_verilog/regincr/test/RegIncr2stage_test.py -v -x --tb=short
% pytest ../tut3_verilog/regincr/test/RegIncr2stage_test.py -v -x --tb=long
```

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

4.9. 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 28 illustrates a random test case for the registered incrementer. We are using the PyMTL3 `Bits` class which provides support for fixed-bitwidth values to ensure a random value that results in modulo arithmetic is handled correctly. 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_test.py` test script, and run the new test case with line tracing enabled as follows:

```
% cd ${TUTROOT}/build
% pytest ../tut3_verilog/regincr/test/RegIncr_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.

```
1  #-------------------------------------------------------------------------
2  # test_random
3  #-------------------------------------------------------------------------
4
5  import random
6
7  def test_random( cmdline_opts ):
8
9    test_vector_table = [( 'in_', 'out*' )]
10   last_result = '?'
11   for i in range(20):
12     rand_value = Bits8( random.randint(0,0xff) )
13     test_vector_table.append( [ rand_value, last_result ] )
14     last_result = Bits8( rand_value + 1, trunc_int=True )
15
16   run_test_vector_sim( RegIncr(), test_vector_table, cmdline_opts )
```

Figure 28: 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.

4.10. 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 29 shows a two-stage registered incrementer that uses structural composition to instantiate and connect two instances of a single-stage registered incrementer. Figure 30 shows the corresponding Verilog module. Line 11 uses a `` `include `` to include the child model that we will be reusing. Notice how we must use the full path (from the root of the project) to the Verilog file we want to include.

Lines 25–31 instantiate the first registered incrementer and lines 35–41 instantiate the second registered incrementer. As mentioned above, we should almost always used named port binding to connect nets to the ports in a module instance. Lines 27–30 illustrate the correct coding convention with one port binding per line and the ports/nets vertically aligned. As shown on lines 26 and 31 the opening and closing parenthesis should be on their own separate lines. We usually declare signals that will be connected to output ports immediately before instantiating the module.

We need to write a new PyMTL3 wrapper for our two-stage registered incrementer, although it will be essentially the same as the wrapper shown in Figure 18 except with a different class name. This illustrates a key point: the PyMTL3 wrapper simply captures the Verilog *interface* and is largely unconcerned with the implementation.

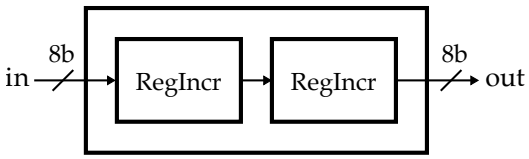

Figure 29: Block Diagram for Two-Stage Registered Incrementer – An eight-bit two-stage registered incrementer that reuses the registered incrementer in Figure 14 through structural composition.

```
1  //========================================================================
2  // RegIncr2stage
3  //========================================================================
4  // Two-stage registered incrementer that uses structural composition to
5  // instantitate and connect two instances of the single-stage registered
6  // incrementer.
7
8  `ifndef TUT3_VERILOG_REGINCR_REG_INCR_2STAGE_V
9  `define TUT3_VERILOG_REGINCR_REG_INCR_2STAGE_V
10
11 `include "tut3_verilog/regincr/RegIncr.v"
12
13 module tut3_verilog_regincr_RegIncr2stage
14 (
15   input  logic       clk,
16   input  logic       reset,
17   input  logic [7:0] in_,
18   output logic [7:0] out
19 );
20
21   // First stage
22
23   logic [7:0] reg_incr_0_out;
24
25   tut3_verilog_regincr_RegIncr reg_incr_0
26   (
27     .clk   (clk),
28     .reset (reset),
29     .in_   (in_),
30     .out   (reg_incr_0_out)
31   );
32
33   // Second stage
34
35   tut3_verilog_regincr_RegIncr reg_incr_1
36   (
37     .clk   (clk),
38     .reset (reset),
39     .in_   (reg_incr_0_out),
40     .out   (out)
41   );
42
43 endmodule
44
45 `endif /* TUT3_VERILOG_REGINCR_REG_INCR_2STAGE_V */
```

Figure 30: Two-Stage Registered Incrementer – An eight-bit two-stage registered incrementer corresponding to Figure 29. This model is implemented using structural composition to instantiate and connect two instances of the single-stage register incrementer.

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

Figure 31: 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.

As always, once we create a new hardware model, we should immediately write a unit test to verify its functionality. Figure 32 shows a test script using test vectors to verify our two-stage registered incrementer. Notice how we must carefully take into account the two-cycle latency of the registered incrementer in order to ensure that each reference output is placed in the correct row of the test vector table. This is because we are using a cycle-by-cycle gray-box testing strategy.

Edit the Verilog source file named `RegIncr2stage.v`. Add lines 33-41 from Figure 30 to instantiate and connect the second stage of the two-stage registered incrementer. Then run all of the test scripts as well as a subset of the test cases as follows:

```
% cd ${TUTROOT}/build
% pytest ../tut3_verilog/regincr/test/RegIncr2stage_test.py -v
```

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

```
% cd ${TUTROOT}/build
% pytest ../tut3_verilog/regincr/test/RegIncr2stage_test.py -k test_small -s
```

The line trace should look similar to what is shown in Figure 31. The line trace in the figure has been annotated to show what each column corresponds to in the model. If you look closely, you can see the input data propagating through both stages of the two-stage registered incrementer. Remember you can generate waveforms for all of the test cases in our new test script as follows:

```
% cd ${TUTROOT}/build
% pytest ../tut3_verilog/regincr/test/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 29. Verify your design by writing a test script that uses test vectors.

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

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

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

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

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

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

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

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

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

import random

def test_random( cmdline_opts ):

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

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

Figure 32: Unit Test Script for Two-Stage Registered Incrementer – A unit test for the two-stage registered incrementer shown in Figure 30 that uses test vectors and the `py.test` unit testing framework.

4.11. Parameterizing a Model with Static Elaboration

To facilitate model reuse and productive design-space exploration, we often want to implement parameterized models. A common example is to parameterize models by the bitwidth of various input and output ports. The registered incrementer in Figure 16 is designed for only 8-bit input values, but we may want to reuse this model in a different context with 4-bit input values or 16-bit input values. We can use Verilog *parameters* to parameterize the port bitwidth for the registered incrementer shown in Figure 16; we would replace references to the constant 7 with a reference to `nbits-1`. Now we can specify the port bitwidth for our register incrementer when we construct the model. We have included a library of parameterized Verilog RTL models in the `vc` subdirectory. Figure 33 shows a combinational incrementer from `vc` that is parameterized by both the port bitwidth and the incrementer amount. The parameters are specified using the special syntax shown on lines 2–5. By convention, we use a `p_` prefix when naming parameters.

Verilog-2001 provides `generate` statements which are meant for static elaboration. Static elaboration happens at compile time, not runtime. We can use static elaboration to *generate* hardware which is fundamentally different from *modeling* hardware. Figure 34 illustrates using `generate` statements to create a multi-stage registered incrementer that is parameterized by the number of stages. The number of stages is specified using the the `p_nstages` parameter shown on line 13. We create a array of signals to hold the intermediate values between stages (line 25), and then we use a `generate` for loop to instantiate and connect the stages. Using `generate` statements is one of the more advanced parts of Verilog, so we will not go into more detail within this tutorial.

Since we want to instantiate the Verilog `RegIncrNstage` model with a `p_nstages` parameter, we cannot directly reuse the previous wrapper for `RegIncr` which does not have parameters. Figure 35 shows now to create a PyMTL3 wrapper for a Verilog module that includes parameters. As long as the names match, the constructor arguments in the PyMTL3 wrapper directly correspond to the Verilog parameters.

One challenge with highly parameterized models is that they can require more complicated verification to test all of the various parameter combinations. The `pytest` framework includes sophisticated support for parameterized testing that can simplify verifying highly parameterized models. Figure 36 shows a test script for the multi-stage registered incrementer model. Because we are using a cycle-by-cycle gray-box testing strategy, the test vectors vary depending on the number of stages. Lines 23–33 define an advanced helper function that takes as input the number of stages and a list

```
1  module vc_Incrementer
2  #(
3    parameter p_nbits     = 1,
4    parameter p_inc_value = 1
5  )(
6    input  logic [p_nbits-1:0] in,
7    output logic [p_nbits-1:0] out
8  );
9
10   assign out = in + p_inc_value;
11
12 endmodule
```

Figure 33: Parameterized Incrementer from `vc` – A combinational incrementer from `vc` that is parameterized by both the port bitwidth and the incrementer amount.

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

`ifndef TUT3_VERILOG_REGINCR_REG_INCR_NSTAGE_V
`define TUT3_VERILOG_REGINCR_REG_INCR_NSTAGE_V

`include "tut3_verilog/regincr/RegIncr.v"

module tut3_verilog_regincr_RegIncrNstage
#(
  parameter p_nstages = 2
)(
  input  logic       clk,
  input  logic       reset,
  input  logic [7:0] in_,
  output logic [7:0] out
);

  // This defines an _array_ of signals. There are p_nstages+1 signals
  // and each signal is 8 bits wide. We will use this array of signals to
  // hold the output of each registered incrementer stage.

  logic [7:0] reg_incr_out [p_nstages+1];

  // Connect the input port of the module to the first signal in the
  // reg_incr_out signal array.

  assign reg_incr_out[0] = in_;

  // Instantiate the registered incrementers and make the connections
  // between them using a generate block.

  genvar i;
  generate
  for ( i = 0; i < p_nstages; i = i + 1 ) begin: gen

    tut3_verilog_regincr_RegIncr reg_incr
    (
      .clk   (clk),
      .reset (reset),
      .in_   (reg_incr_out[i]),
      .out   (reg_incr_out[i+1])
    );

  end
  endgenerate

  // Connect the last signal in the reg_incr_out signal array to the
  // output port of the module.

  assign out = reg_incr_out[p_nstages];

endmodule

`endif /* TUT3_VERILOG_REGINCR_REG_INCR_NSTAGE_V */
```

Figure 34: N-Stage Registered Incrementer – A parameterized registered incrementer where the number of stages is specified using a Verilog parameter.

```
1  #=========================================================================
2  # RegIncrNstage
3  #=========================================================================
4  # Registered incrementer that is parameterized by the number of stages.
5
6  from pymtl3 import *
7  from pymtl3.passes.backends.verilog import *
8
9  class RegIncrNstage( VerilogPlaceholder, Component ):
10   def construct( s, p_nstages=2 ):
11     s.in_ = InPort ( 8 )
12     s.out = OutPort( 8 )
```

Figure 35: Registered Incrementer Wrapper Parameterized by the Number of Stages – PyMTL3 wrapper for the `RegIncrNstage` Verilog module.

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

On lines 39–55, we use `pytest.mark.parametrize` to succinctly generate eight test cases that test both two- and three-stage registered incrementers with small, large, overflow, and random input values. We use another helper function (named `mk_test_case_table`) which is provided by the PyMTL3 framework to create a test case table. A test case table compactly represents a set of test cases. Each row corresponds to a test case, and the first column is always the name of the test case. The remaining columns correspond to the test parameters. The first row of the test case table is always a special "header string" that specifies the name of each test parameter. In this example, there are two test parameters: the number of stages (`nstages`) and the test inputs (`inputs`). Notice how we use the `sample` function from the standard Python `random` module to generate a random sequence of input values. The `mk_test_case_table` creates a data structure suitable for passing into `pytest.mark.parametrize`. For technical reasons, we need to use the `**` operator to pass this data structure into `pytest.mark.parametrize`, as shown on line 50. The test function on lines 51–55 includes a `test_params` argument that will contain the test parameters corresponding to one row of the test case table. On lines 52–53, we read these test parameters, and then on lines 54–55 we use the `run_test_vector_sim` and the `mk_test_vector_table` helper functions to actually run a test.

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

Let's run all of the test cases for our multi-stage registered incrementer.

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

import collections
import pytest

from random import sample, seed

from pymtl3 import *

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

# To ensure reproducible testing

seed(0xdeadbeef)

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

def mk_test_vector_table( nstages, inputs ):

  inputs.extend( [0]*nstages )

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

  return test_vector_table

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

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

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

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

Figure 36: Unit Test Script for Parameterized Registered Incrementer – A unit test for the parameterized registered incrementer shown in Figure 34.

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

../tut3_verilog/regincr/test/RegIncrNstage_test.py::test[2stage_small] PASSED
../tut3_verilog/regincr/test/RegIncrNstage_test.py::test[2stage_large] PASSED
../tut3_verilog/regincr/test/RegIncrNstage_test.py::test[2stage_overflow] PASSED
../tut3_verilog/regincr/test/RegIncrNstage_test.py::test[2stage_random] PASSED
../tut3_verilog/regincr/test/RegIncrNstage_test.py::test[3stage_small] PASSED
../tut3_verilog/regincr/test/RegIncrNstage_test.py::test[3stage_large] PASSED
../tut3_verilog/regincr/test/RegIncrNstage_test.py::test[3stage_overflow] PASSED
../tut3_verilog/regincr/test/RegIncrNstage_test.py::test[3stage_random] PASSED
../tut3_verilog/regincr/test/RegIncrNstage_test.py::test_random[1] PASSED
../tut3_verilog/regincr/test/RegIncrNstage_test.py::test_random[2] PASSED
../tut3_verilog/regincr/test/RegIncrNstage_test.py::test_random[3] PASSED
../tut3_verilog/regincr/test/RegIncrNstage_test.py::test_random[4] PASSED
../tut3_verilog/regincr/test/RegIncrNstage_test.py::test_random[5] PASSED
../tut3_verilog/regincr/test/RegIncrNstage_test.py::test_random[6] PASSED

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

Figure 37: `pytest` **Parameterized Output –** Each line corresponds to one test case. Test cases generated using `pytest.mark.parametrize` use square brackets to denote each generated test case.

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

The output should look similar to what is shown in Figure 37. Notice how the `pytest` framework names the generated test cases. When using a test case table, the `pytest` framework puts the test case name in square brackets after the test function name (e.g., `test[2stage_small]`). When not using a test case table, the `pytest` framework uses the arguments to the test function in square brackets after the test function name (e.g., `test_random[2]`).

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

```
% cd ${TUTROOT}/build
% pytest ../tut3_pymtl/regincr/test/RegIncrNstage_test.py -k 3stage -sv
```

★ *To-Do On Your Own:* Parameterize the input/output port bitwidth for the basic registered incrementer in Figure 16. 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.

5. Sort Unit

The previous section introduces the key Verilog concepts and primitives that we will use to implement more complex RTL models including: declaring a port-based module interface; declaring internal state and wires using `logic` variables; declaring `always @( posedge clk )` concurrent blocks to model logic that executes on every rising clock edge; declaring `always @(*)` concurrent blocks to model combinational logic that executes one or more times within a clock cycle; and creating PyMTL3 wrappers. 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 an RTL model. We will also learn how to use a simulator to evaluate a design. Most of the code for this section is provided for you in the `tut3_verilog/sort` subdirectory.

5.1. FL Model of Sort Unit

We begin by designing an FL model of our target sort unit. *Keep in mind that we will almost always provide students with an appropriate FL model. Students will not need to develop their own FL models from scratch!* Recall that FL models implement the *functionality* but not the timing of the hardware target. Figure 38 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 39 shows how to implement an FL model for the sort unit in PyMTL3. On lines 17 and 20, we use Python list comprehensions to create lists of four input and output ports. On lines 32 and 36, we use the standard Python `map` function to easily convert all input/output values into strings for line tracing. Notice how our line tracing code checks the input/output valid bit, and if the input/output is invalid then we clear the corresponding string to all spaces. This means the line trace will show spaces when the input/output values are invalid, but the line trace is still always a fixed width to ensure the columns stay aligned. We generally use this idea of displaying spaces in the line trace when "nothing is happening"; this makes it easy to see true activity in the line trace.

The `update_ff` concurrent block on lines 22–26 defines the actual functional-level behavior. There are many kinds of FL models, and here we create an FL model by using RTL interfaces but "magic" sorting. The `update` concurrent block in our sort unit FL model uses the 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.

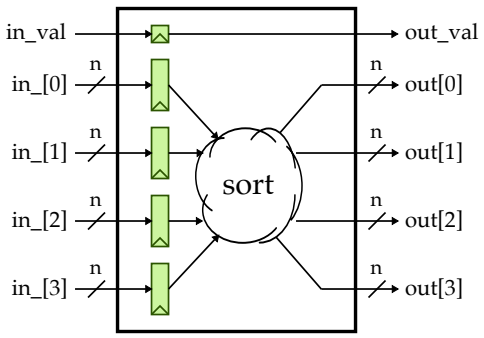

Figure 38: Cloud Diagram for Sort Unit FL Model – Cloud diagrams use "clouds" to abstractly represent logic without worry about the actual implementation details. The sort unit FL model takes four input values and sorts them such that the `out[0]` port has the smallest value and the `out[3]` port has the largest value. Input/output valid bits indicate when the input/output values are valid.

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

from copy import deepcopy
from pymtl3 import *

class SortUnitFL( Component ):

  # Constructor

  def construct( s, p_nbits=8 ):

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

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

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

  # Line tracing

  def line_trace( s ):

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

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

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

Figure 39: Sort Unit FL Model – FL model of four-element sort unit corresponding to Figure 38.

Notice that although this model in no way attempts to capture any timing of the hardware target, it is still a "single-cycle" model. This is due to the PyMTL3 semantics of `update_ff` concurrent blocks and non-blocking assignments, and this is why we show input registers in the cloud diagram in Figure 38. Although it is also possible to implement FL models using `update` concurrent blocks, we have found using `update_ff` concurrent blocks to be significantly easier. Using `update` concurrent blocks means the block can be called multiple times in a cycle, increases the likelihood of creating combinational loops when composing FL models, and complicates incrementally refining an FL model into an RTL model.

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

The PyMTL3 model is in `SortUnitFL.py` and the corresponding test script is in `SortUnitFL_test.py`. This test script uses test vector tables similar in spirit to the unit testing for the registered incrementer in Figure 32. The test script for `SortUnitFL` includes a variety of directed and random test cases. Note that we usually try to ensure that the very first test case is always the simplest possible test case we can imagine. For this model, our first test case simply sorts a single set of four input values. You can run all of the tests and display the line trace for the basic test case as follows:

```
% cd ${TUTROOT}/build
% pytest ../tut3_verilog/sort/test/SortUnitFL_test.py -v
% pytest ../tut3_verilog/sort/test/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 RTL model. Using the FL model to write tests also ensures if the RTL model fails a test, it is more likely due to the 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. Flat RTL Model of Sort Unit

Now that we have a behavioral FL model, we can develop an RTL model. Recall that RTL models are *cycle-accurate*, *resource-accurate*, and *bit-accurate* representations of hardware. Although RTL models are usually more tedious to construct, they are also the most accurate with respect to the target hardware. Figure 40 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

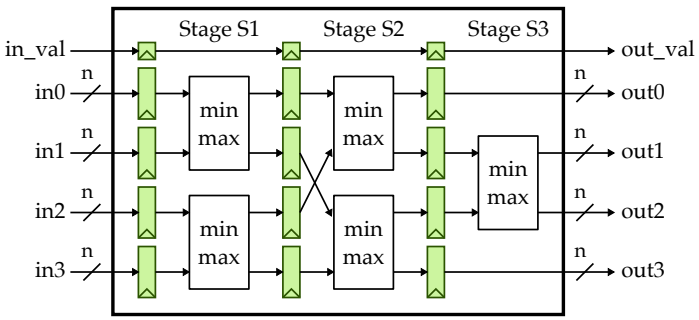

Figure 40: Block Diagram for Sort Unit RTL Model – The RTL model implements a three-stage pipelined, bitonic sorting network.

```
1  module tut3_verilog_sort_SortUnitFlat
2  #(
3    parameter p_nbits = 1
4  )(
5    input  logic               clk,
6    input  logic               reset,
7
8    input  logic               in_val,
9    input  logic [p_nbits-1:0] in0,
10   input  logic [p_nbits-1:0] in1,
11   input  logic [p_nbits-1:0] in2,
12   input  logic [p_nbits-1:0] in3,
13
14   output logic               out_val,
15   output logic [p_nbits-1:0] out0,
16   output logic [p_nbits-1:0] out1,
17   output logic [p_nbits-1:0] out2,
18   output logic [p_nbits-1:0] out3
19 );
```

Figure 41: Interface for the Four-Element Sorter – The interface corresponds to the diagram in Figure 40 and is parameterized by the bitwidth of each element.

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 41 illustrates the Verilog code that describes the interface for the sort unit. Notice how we have parameterized the interface by the bitwidth of each element. Lines 2–4 declare a parameter named `p_nbits` and give it a default value of one bit. We use this parameter when declaring the bitwidth of the input and output ports, and we will also use this parameter in the implementation.

Figure 42 shows how to implement a flat RTL model for the sort unit in Verilog. 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 `always_comb` blocks) from the combinational logic (modeled with `always_ff` 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 must explicitly reset state. Line 31 uses the `reset` signal to reset the valid bit register to zero in the first stage of the pipeline. Lines 36–47 correspond to the first stage in Figure 40 with two min/max units.

The RTL model is in `SortUnitFlat.v`, the PyMTL3 wrapper is in `SortUnitFlat.py`, and the corresponding test script is in `SortUnitFlat_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 `SortUnitFL_test.py`. This ability to share test vectors, cases, and/or harnesses across many different test scripts is a significant benefit of the `pytest` framework. You can run all of the tests and display the line trace for the basic test case as follows:

```
% cd ${TUTROOT}/build
% pytest ../tut3_verilog/sort/test/SortUnitFlat_test.py -v
% pytest ../tut3_verilog/sort/test/SortUnitFlat_test.py -k test_basic -s
```

The line trace for the sort unit RTL model is shown in Figure 43. Cycle 1 and cycle 2 are during the reset phase. Cycle 3 doesn't have a valid input. On cycle 4, there is a valid set of four input

```
1  //------------------------------------------------------------------------
2  // Stage S0->S1 pipeline registers
3  //------------------------------------------------------------------------
4
5  logic               val_S1;
6  logic [p_nbits-1:0] elm0_S1;
7  logic [p_nbits-1:0] elm1_S1;
8  logic [p_nbits-1:0] elm2_S1;
9  logic [p_nbits-1:0] elm3_S1;
10
11 always_ff @( posedge clk ) begin
12   val_S1  <= (reset) ? 0 : in_val;
13   elm0_S1 <= in0;
14   elm1_S1 <= in1;
15   elm2_S1 <= in2;
16   elm3_S1 <= in3;
17 end
18
19 //------------------------------------------------------------------------
20 // Stage S1 combinational logic
21 //------------------------------------------------------------------------
22 // Note that we explicitly catch the case where the elements contain
23 // X's and propagate X's appropriately. We would not need to do this if
24 // we used a continuous assignment statement with a ternary conditional
25 // operator.
26
27 logic [p_nbits-1:0] elm0_next_S1;
28 logic [p_nbits-1:0] elm1_next_S1;
29 logic [p_nbits-1:0] elm2_next_S1;
30 logic [p_nbits-1:0] elm3_next_S1;
31
32 always_comb begin
33
34   // Sort elms 0 and 1
35
36   if ( elm0_S1 <= elm1_S1 ) begin
37     elm0_next_S1 = elm0_S1;
38     elm1_next_S1 = elm1_S1;
39   end
40   else if ( elm0_S1 > elm1_S1 ) begin
41     elm0_next_S1 = elm1_S1;
42     elm1_next_S1 = elm0_S1;
43   end
44   else begin
45     elm0_next_S1 = 'x;
46     elm1_next_S1 = 'x;
47   end
48
49   // Sort elms 2 and 3
50
51   if ( elm2_S1 <= elm3_S1 ) begin
52     elm2_next_S1 = elm2_S1;
53     elm3_next_S1 = elm3_S1;
54   end
55   else if ( elm2_S1 > elm3_S1 ) begin
56     elm2_next_S1 = elm3_S1;
57     elm3_next_S1 = elm2_S1;
58   end
59   else begin
60     elm2_next_S1 = 'x;
61     elm3_next_S1 = 'x;
62   end
63
64 end
```

Figure 42: First Stage of the Flat Sorter Implementation – First pipeline stage of the sorter using a flat implementation corresponding to the diagram in Figure 40.

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

Figure 43: 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. Each line corresponds to one (and only one!) cycle, and the fixed-width columns correspond to either the state at the beginning of the corresponding cycle or the output of combinational logic during that cycle. If the valid bit is not set, then the corresponding list of values is not shown.

values available on the input ports, and on cycle 5, we can see that this set of four values is now in the first set of pipeline registers. Recall that our line trace shows the state at the beginning of the corresponding cycle. During cycle 5, pipeline stage S1 swaps elements 0 and 1, and also swaps elements 2 and 3. We can see the result of these swaps by looking at the four values on cycle 5 at the beginning of pipeline stage S2. During cycle 6, pipeline stage S2 swaps elements 0 and 2, and also swaps elements 1 and 3. During cycle 7, pipeline stage S1 swaps elements 1 and 2 before writing the results to the output ports.

★ *To-Do On Your Own:* Make a copy of the sort unit implementation file so you can put things back to the way they were when you are finished. The sort unit currently sorts the four input numbers from smallest to largest. Change to the sort unit 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]
```

5.3. Structural RTL Model of Sort Unit

The flat implementation in `SortUnitFlat.v` is complex and monolithic and it fails to really exploit the structure inherent in the sort unit. 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 44 shows how to implement a structural RTL model for the sort unit in Verilog. 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. Our design instantiates three kinds of modules: `vc_ResetReg`, `vc_Reg`, and `tut3_verilog_sort_MinMaxUnit`. The register modules are provided in the VC library. Notice how we still use the parameter `p_nbits` to declare various internal variables, but in addition, we use this parameter when instantiating parameterized sub-modules. For example, the `vc_Reg` module is parameterized, and this allows us to easily create pipeline registers of any

```
//------------------------------------------------------------------------
// Stage S0->S1 pipeline registers
//------------------------------------------------------------------------

logic val_S1;

vc_ResetReg#(1) val_S0S1
(
 .clk   (clk),
 .reset (reset),
 .d     (in_val),
 .q     (val_S1)
);

// This is probably the only place where it might be acceptable to use
// positional port binding since (a) it is so common and (b) there are
// very few ports to bind.

logic [p_nbits-1:0] elm0_S1;
logic [p_nbits-1:0] elm1_S1;
logic [p_nbits-1:0] elm2_S1;
logic [p_nbits-1:0] elm3_S1;

vc_Reg#(p_nbits) elm0_S0S1( clk, elm0_S1, in0 );
vc_Reg#(p_nbits) elm1_S0S1( clk, elm1_S1, in1 );
vc_Reg#(p_nbits) elm2_S0S1( clk, elm2_S1, in2 );
vc_Reg#(p_nbits) elm3_S0S1( clk, elm3_S1, in3 );

//------------------------------------------------------------------------
// Stage S1 combinational logic
//------------------------------------------------------------------------

logic [p_nbits-1:0] mmuA_out_min_S1;
logic [p_nbits-1:0] mmuA_out_max_S1;

tut3_verilog_sort_MinMaxUnit#(p_nbits) mmuA_S1
(
  .in0     (elm0_S1),
  .in1     (elm1_S1),
  .out_min (mmuA_out_min_S1),
  .out_max (mmuA_out_max_S1)
);

logic [p_nbits-1:0] mmuB_out_min_S1;
logic [p_nbits-1:0] mmuB_out_max_S1;

tut3_verilog_sort_MinMaxUnit#(p_nbits) mmuB_S1
(
  .in0     (elm2_S1),
  .in1     (elm3_S1),
  .out_min (mmuB_out_min_S1),
  .out_max (mmuB_out_max_S1)
);
```

Figure 44: First Stage of the Structural Sorter Implementation – First pipeline stage of the sorter using a structural implementation corresponding to the diagram in Figure 40.

bitwidth. Even though we are using a structural implementation strategy, we still cleanly separate the sequential logic from the combinational logic. We still use comments and explicit suffixes to make it clear what pipeline stage we are modeling.

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

```
% cd ${TUTROOT}/build
% pytest ../tut3_verilog/sort/test/SortUnitStruct_test.py -v
% pytest ../tut3_verilog/sort/test/SortUnitStruct_test.py -k test_basic -s
```

The structural implementation is incomplete because the actual implementation of the min/max unit in `MinMaxUnit.v` is not finished. You should go ahead and implement the min/max unit, and then *as always you should write a unit test to verify the functionality of your MinMax unit!* Add some line tracing for the min/max unit. You should have enough experience based on the previous sections to be able to create a unit test from scratch and run it using `pytest`. You should name the new test script `MinMaxUnit_test.py`. You can use the registered incrementer model as an example for both implementing the min/max unit and for writing the corresponding test script. Note that the line trace for the sort unit structural RTL model should be the same as in Figure 43, since these are really just two different implementations of the sort unit RTL.

5.4. Evaluating the 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 45. 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 16–19 define a standard Python dictionary that maps strings to model types. Then on line 21, we can simply use this dictionary to instantiate the correct model based on the `--impl` command line option. The simulator will take care of conditionally generating waveforms based on the `--dump-vcd` command line option by crafting a dictionary to pass to a `stdlib` test utility function. Line 33 turn on line tracing based on the `--trace` command line option. The main simulator loop on lines 40–59 iterates through the input dataset and sets the corresponding input ports. The simulator loops keeps a counter to track how many valid outputs have been received, and thus to determine when to stop the simulation. A key difference between a simulator and a unit test is that the simulator should also report various statistics that help us evaluate our design. The `--stats` command line option will display the number of cycles to finish processing the input dataset, and the average number of cycles per sort. You can run the simulator script for the two sort unit RTL models as follows:

```
% cd ${TUTROOT}/build
% ../tut3_verilog/sort/sort-sim --stats --impl rtl-flat
```

```
opts = parse_cmdline()

# Create input datasets

ninputs = 100
inputs  = []

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

...

# Instantiate and elaborate the design

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

model = model_impl_dict[ opts.impl ]()

...

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

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

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

model.sim_reset()

# Tick simulator until evaluation is finished

counter = 0
while counter < ninputs:

  if model.out_val:
    counter += 1

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

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

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

  model.sim_tick()

# Report various statistics

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

Figure 45: 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.

```
% ../tut3_verilog/sort/sort-sim --stats --impl rtl-struct
```

Not surprisingly, it should take one cycle on average since our RTL model is fully pipelined. 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_verilog/sort/sort-sim --stats --impl rtl-flat --input random
% ../tut3_verilog/sort/sort-sim --stats --impl rtl-flat --input sorted-fwd
% ../tut3_verilog/sort/sort-sim --stats --impl rtl-flat --input sorted-rev
```

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

```
% cd ${TUTROOT}/build
% ../tut3_verilog/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.

6. Greatest Common Divisor

The previous section introduced the process of refining a design from an initial FL model to an RTL model. In this section, we will apply what we have learned to study a more complicated hardware unit that calculates the greatest common divisor (GCD) of two input operands. We will gain experience with latency-insensitive stream interfaces that implement a val/rdy microprotocol, unit testing with stream sources/sinks, and using a control/datapath split to implement RTL models. The code for this section is provided for you in the `tut3_verilog/gcd` subdirectory.

6.1. FL Model of GCD Unit

As before, we begin by designing an FL model of our target GCD unit. *Keep in mind that we will almost always provide students with an appropriate FL model. Students will not need to develop their own FL models from scratch!* Even so, it can be useful to understand how these FL models work. Figure 46 shows a cloud diagram for the GCD unit FL model, and Figure 47 shows the approach we will use in our RTL models based on Euclid's algorithm. The GCD unit will take two 16-bit operands and produce a 16-bit result. A key feature of the GCD unit is its use of latency-insensitive stream interfaces to manage flow control for the requests and responses. The interface for the registered incrementer in Section 4 included no extra control signals. A module that wants to use the registered incrementer must explicitly handle the fact that the unit always takes exactly one cycle. The interface for the sort

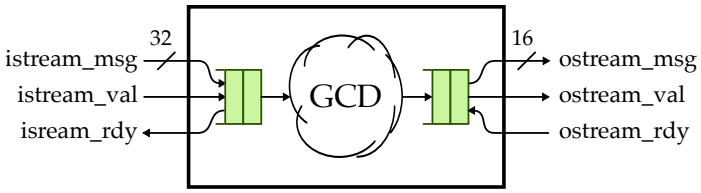

Figure 46: Functional-Level Implementation of GCD Unit – Input and output use latency-insensitive stream/rdy interfaces. The input message includes two 16-bit operands; output message is an 16-bit result. Clock and reset signals are not shown.

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

Figure 47: Euclid's GCD Algorithm – Iteratively subtract the smaller value from the larger value until one of them is zero, at which time the GCD is the non-zero value. This is executable Python code.

unit in Section 5 included an extra valid signal. A module that wants to use the sort unit could be carefully constructed so as to be agnostic to the latency of the sort unit; this would enable flexibly trying out different sorting algorithms each with different latencies by using the valid signal to know when the result is valid. One issue with including just a valid signal is that there is no way to know if the sort unit is busy, and there is no way to tell the sort unit that we are not ready to accept the result. In other words, there is no provision for *back pressure*. As shown in Figure 46, our GCD design will use a fully latency-insensitive stream interface implemented through a val/rdy microprotocol which involves an extra valid signal and ready signal. These signals will allow additional flexibility: the GCD unit can indicate it is not ready to accept a new GCD input, and another module can indicate that it is not ready to accept the GCD output.

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

Based on the discussion so far, the benefit of a latency-insensitive stream 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 these latency-insensitive stream interfaces extensively throughout the course.

In Figure 46, we can see that we often use stream queue adapters to simplify designing FL models that interact with stream interfaces. Figure 48 shows how to implement an FL model for the GCD unit in PyMTL3. The actual work of the FL model takes place on line 40. We use the `gcd` function from

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

Figure 48: Gcd Unit FL Model – FL model of greatest-common divisor unit corresponding to Figure 46.

the standard Python `math` module to calculate the GCD of the two input operands. This example illustrates two important PyMTL3 features: interfaces and interface adapters.

Lines 23–24 of Figure 48 use interfaces instead of ports as the interface for our GCD unit. An RTL interface 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 `IStreamIfc` and `OStreamIfc` from `stdlib.stream` which implement the aforementioned val/rdy microprotocol. This RTL interface groups together the valid, ready, and message ports.

```
1  #-------------------------------------------------------------------------
2  # TestHarness
3  #-------------------------------------------------------------------------
4
5  class TestHarness( Component ):
6
7    def construct( s, gcd ):
8
9      # Instantiate models
10
11     s.src  = StreamSourceFL( Bits32 )
12     s.sink = StreamSinkFL( Bits16 )
13     s.gcd  = gcd
14
15     # Connect
16
17     s.src.ostream //= s.gcd.istream
18     s.gcd.ostream //= s.sink.istream
19
20   def done( s ):
21     return s.src.done() and s.sink.done()
22
23   def line_trace( s ):
24     return s.src.line_trace() + " > " + \
25            s.gcd.line_trace() + " > " + \
26            s.sink.line_trace()
```

Figure 49: Excerpt from Unit Test Script for GCD Unit FL Model – Latency insensitive interfaces enable us to use generic sources and sinks for testing.

Lines 28–29 of Figure 48 instantiate two stream queue adapters provided by the PyMTL3 standard library. Interface adapters take the data type as constructor arguments, and then enable the logic within the component to interact with these ports through methods. In this example, we are using `IStreamDeqAdapterFL` and `OStreamEnqAdapterFL` from `stdlib.stream`. A `IStreamDeqAdapterFL` connects to an `IStreamIfc` and provides a standard Python `deq` method for the FL model to use. A `OStreamEnqAdapterFL` connects to an `OStreamIfc` and provides a standard Python `enq` method for the FL model to use. We also make use of the `update_once` blocks that are meant to be only called once in each clock cycle. `update_once` blocks are supposed to call methods and modify signals using `@=` blocking assignments. The `update_once` block first invokes `deq.rdy()` method to check if the input stream queue adapter has a message to pop and `enq.rdy()` to check if the output stream queue adapter has available slots to accept a message. If both are ready, we can dequeue a message from the input stream queue adapter (line 39) and enqueue the result using the output stream queue adapter (line 40). These queue adapters significantly simplify implementing FL models, since we only need to implement the functionality using method calls without explicitly managing the low-level RTL ports.

The PyMTL3 FL model is in `GcdUnitFL.py` and the corresponding test script is in `GcdUnitFL_test.py` in the `test` subdirectory. One of the nice features of using a latency-insensitive stream interface is that it enables us to use a common framework for sending messages into the device-under-test (DUT) and then verifying that the correct messages come out of the DUT. `stdlib.stream` includes the `StreamSourceFL` and `StreamSinkFL` models for this purpose. Figure 49 illustrates the test harness included in the GCD unit test script. We instantiate a test source and attach it to the GCD unit's input stream interface, and then we instantiate a test sink and attach it to the GCD unit's output stream

Figure 50: Verifying GCD Using Stream Sources and Sinks – Parameterized stream sources send messages over a stream interface, and parameterized stream sinks receive messages over a stream interface and compare each message to a previously specified reference message.

interface. Figure 50 illustrates the overall connectivity in the test harness. Notice how interfaces enable us to connect three ports with a single connect statement (lines 17–18). The test source includes the ability to delay the messages going into the DUT and the test sink includes the ability to apply back-pressure to the DUT. More specifically we can set an *initial delay* (i.e., how many cycles to delay a message after reset) and an *interval delay* (i.e., how many cycles after receiving a message to delay the next message). By using various combinations of these delays we can more robustly ensure that our flow-control logic is working correctly. Note that these test cases illustrate both *directed black-box* and *randomized black-box* testing strategies. The test cases are black-box since they do not depend on the timing within the DUT.

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

```
% cd ${TUTROOT}/build
% pytest ../tut3_verilog/gcd/test/GcdUnitFL_test.py -k basic_0x0 -s
```

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

```
% cd ${TUTROOT}/build
% pytest ../tut3_verilog/gcd/test/GcdUnitFL_test.py -k basic -s
```

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

```
% cd ${TUTROOT}/build
% pytest ../tut3_verilog/gcd/test/GcdUnitFL_test.py -k random -s
```

Figure 51 illustrates a portion of the line trace for the randomized testing. Notice that the line trace tells something about what is going on with each val/rdy interface. A period (`.`) indicates that the interface is not ready but also not valid; a hash (`#`) indicates that the interface is valid but not ready; a space indicates that the interface is ready but not valid. The actual message is displayed when it is transferred from the producer to the consumer. We can see a message being sent from the test source into the GCD unit on cycle 7 and although the result is valid on cycle 8 the test sink is not ready until cycle 13 to accept the result. On cycles 8–10 the test source does not have a new message to send to the GCD unit. On cycle 12 it does indeed have a new message, but the GCD unit is not ready because

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

Figure 51: 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.

it is still waiting on the test sink. Finally, on cycle 13 the test sink is ready and the GCD unit is able to send the result and accept a new input.

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

6.2. RTL Model of GCD Unit

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

Take a look at the datapath interface which is also shown in Figure 54. We cleanly identify the data signals, control signals, and status signals. Figure 54 also shows the first two datapath components, but take a look in `GcdUnit.v` to see the entire datapath. Notice how we use a very structural implementation that *exactly* matches the datapath diagram in Figure 52. We leverage several modules from the VC library (e.g., `vc_Mux2`, `vc_ZeroComparator`, `vc_Subtractor`). You should use a similar structural approach when building your own datapaths for this course. For a net that moves data from left to right in the datapath diagram, we usually declare a dedicated wire right before the module instance (e.g., `a_mux_out` and `a_reg_out`). 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., `b_reg_out` and `b_sub_out`).

Take a look at the control unit and notice the stylized way we write FSMs. An FSM-based control unit should have three parts: a sequential concurrent block for the state, a combinational concurrent block for the state transitions, and a combinational concurrent block for the state outputs. We often

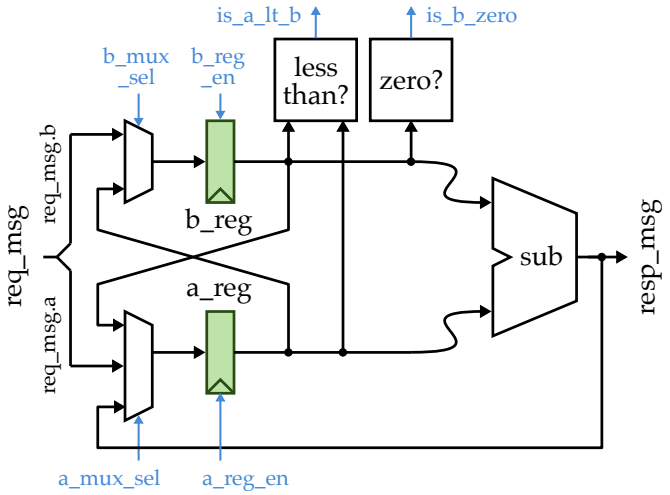

Figure 52: Datapath Diagram for GCD – Datapath includes two state registers and required muxing and arithmetic units to iteratively implement Euclid's algorithm.

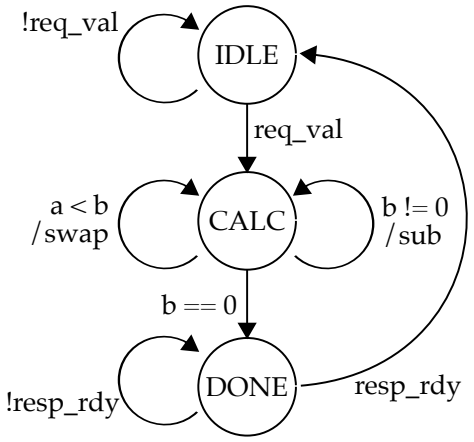

Figure 53: FSM Diagram for GCD – A hybrid Moore/Mealy FSM for controlling the datapath in Figure 52. Mealy transitions in the calc state determine whether to swap or subtract.

use case statements to compactly represent the state transitions and outputs. Figure 55 shows the portion of the FSM responsible for setting the output signals. We use a Verilog function or a Verilog task to set all of the control signals in a single line. In general, you should prefer Verilog functions vs. Verilog tasks since Verilog tasks can include non-synthesizable constructs while Verilog functions are much more likely to be synthesizable. Essentially, we have created a "control signal table" in our Verilog code which exactly matches what we might draw on a piece of paper. There is one row for each state or Mealy transition and one column for each control signal. These compact control signal tables simplify coding complicated FSMs (or indeed other kinds of control logic) and can enable a designer to quickly catch bugs (e.g., are the enable signals always set to either zero or one?).

Figure 56 shows a portion of the top-level module that connects the datapath and control unit together. Lines 35 and 42 use the new implicit connection operator (`.*`) provided by SystemVerilog. Using the implicit connection operator during module instantiation means to connect every port to a signal with the same name. So these two lines take care of connecting all of the control and status signals. This is a powerful way to write more compact code which avoids connectivity bugs, especially when connecting the datapath and control unit.

```
module tut3_verilog_gcd_GcdUnitDpath
(
  input  logic        clk,
  input  logic        reset,

  // Data signals

  input  logic [31:0] istream_msg,
  output logic [15:0] ostream_msg,

  // Control signals

  input  logic        a_reg_en,   // Enable for A register
  input  logic        b_reg_en,   // Enable for B register
  input  logic [1:0]  a_mux_sel,  // Sel for mux in front of A reg
  input  logic        b_mux_sel,  // sel for mux in front of B reg

  // Status signals

  output logic        is_b_zero,  // Output of zero comparator
  output logic        is_a_lt_b   // Output of less-than comparator
);

  localparam c_nbits = 16;

  // Split out the a and b operands

  logic [c_nbits-1:0] istream_msg_a;
  assign istream_msg_a = istream_msg[31:16];

  logic [c_nbits-1:0] istream_msg_b;
  assign istream_msg_b = istream_msg[15:0];

  // A Mux

  logic [c_nbits-1:0] b_reg_out;
  logic [c_nbits-1:0] sub_out;
  logic [c_nbits-1:0] a_mux_out;

  vc_Mux3#(c_nbits) a_mux
  (
    .sel   (a_mux_sel),
    .in0   (istream_msg_a),
    .in1   (b_reg_out),
    .in2   (sub_out),
    .out   (a_mux_out)
  );

  // A register

  logic [c_nbits-1:0] a_reg_out;

  vc_EnReg#(c_nbits) a_reg
  (
    .clk   (clk),
    .reset (reset),
    .en    (a_reg_en),
    .d     (a_mux_out),
    .q     (a_reg_out)
  );
```

Figure 54: Portion of GCD Datapath Unit – We use `struct` types to encapsulate both control and status signals and we use a preprocessor macro from the GCD message struct to determine how to size the datapath components.

```
1  //------------------------------------------------------------------------
2  // State Outputs
3  //------------------------------------------------------------------------
4
5  localparam a_x   = 2'dx;
6  localparam a_ld  = 2'd0;
7  localparam a_b   = 2'd1;
8  localparam a_sub = 2'd2;
9
10 localparam b_x   = 1'dx;
11 localparam b_ld  = 1'd0;
12 localparam b_a   = 1'd1;
13
14 function void cs
15 (
16   input logic       cs_istream_rdy,
17   input logic       cs_ostream_val,
18   input logic [1:0] cs_a_mux_sel,
19   input logic       cs_a_reg_en,
20   input logic       cs_b_mux_sel,
21   input logic       cs_b_reg_en
22 );
23 begin
24   istream_rdy = cs_istream_rdy;
25   ostream_val = cs_ostream_val;
26   a_reg_en    = cs_a_reg_en;
27   b_reg_en    = cs_b_reg_en;
28   a_mux_sel   = cs_a_mux_sel;
29   b_mux_sel   = cs_b_mux_sel;
30 end
31 endfunction
32
33 // Labels for Mealy transistions
34
35 logic do_swap;
36 logic do_sub;
37
38 assign do_swap = is_a_lt_b;
39 assign do_sub  = !is_b_zero;
40
41 // Set outputs using a control signal "table"
42
43 always_comb begin
44
45   set_cs( 0, 0, a_x, 0, b_x, 0 );
46   case ( state_reg )
47     //                            recv send a mux  a  b mux b
48     //                            rdy  val  sel    en sel   en
49     STATE_IDLE:               cs( 1,   0,   a_ld,  1, b_ld, 1 );
50     STATE_CALC: if ( do_swap ) cs( 0,   0,   a_b,   1, b_a,  1 );
51            else if ( do_sub  ) cs( 0,   0,   a_sub, 1, b_x,  0 );
52     STATE_DONE:               cs( 0,   1,   a_x,   0, b_x,  0 );
53     default                   cs('x,  'x,   a_x,  'x, b_x, 'x );
54
55   endcase
56
57 end
```

Figure 55: Portion of GCD FSM-based Control Unit for State Outputs – We can use a task to create a "control signal table" with one row per state or Mealy transition and one column per control signal. Local parameters can help compactly encode various control signal values.

```
1  module tut3_verilog_gcd_GcdUnitRTL
2  (
3    input  logic        clk,
4    input  logic        reset,
5
6    input  logic        istream_val,
7    output logic        istream_rdy,
8    input  logic [31:0] istream_msg,
9
10   output logic        ostream_val,
11   input  logic        ostream_rdy,
12   output logic [15:0] ostream_msg
13 );
14
15   //------------------------------------------------------------------------
16   // Connect Control Unit and Datapath
17   //------------------------------------------------------------------------
18
19   // Control signals
20
21   logic        a_reg_en;
22   logic        b_reg_en;
23   logic [1:0]  a_mux_sel;
24   logic        b_mux_sel;
25
26   // Data signals
27
28   logic        is_b_zero;
29   logic        is_a_lt_b;
30
31   // Control unit
32
33   tut3_verilog_gcd_GcdUnitCtrl ctrl
34   (
35     .*
36   );
37
38   // Datapath
39
40   tut3_verilog_gcd_GcdUnitDpath dpath
41   (
42     .*
43   );
44
45 endmodule
```

Figure 56: Portion of GCD Top-Level Module – We use the new implicit connection operator (`.*`) to automatically connect all of the control and status signals to both the control unit and datapath.

```
1  #=========================================================================
2  # GCD Unit RTL Model
3  #=========================================================================
4
5  from pymtl3 import *
6  from pymtl3.passes.backends.verilog import *
7  from pymtl3.stdlib.stream import IStreamIfc, OStreamIfc
8
9  class GcdUnit( VerilogPlaceholder, Component ):
10   def construct( s ):
11     s.istream = IStreamIfc( Bits32 )
12     s.ostream = OStreamIfc( Bits16 )
```

Figure 57: GCD Unit Wrapper – PyMTL3 wrapper for the Verilog RTL implementation of the gcd unit.

Figure 57 shows how we can use PyMTL3 to create a Python wrapper for the GCD unit. We can use the PyMTL3 `IStreamIfc` and `OStreamIfc` interfaces which directly correspond to the val/rdy/msg ports in the Verilog RTL. The test script is in `GcdUnit_test.py`. The RTL model is able use the exact same test setup as the GCD unit FL model, even though the FL and RTL models all take different amounts of time to calculate the GCD. This illustrates the power of using latency-insensitive stream interfaces. We can run all of the tests and display the line trace for the basic test case with delays in the test sink like this:

```
% cd ${TUTROOT}/build
% pytest ../tut3_verilog/gcd/test/GcdUnit_test.py -v
% pytest ../tut3_verilog/gcd/test/GcdUnit_test.py -sv -k basic_0x0
```

Figure 58 illustrates a portion of the line trace for the randomized testing. 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 296. The GCD unit is in the idle state and transitions into the calc state. It does five subtractions and a final swap before transitioning into the done state on cycle 304. The result is valid but the test sink is not ready, so the GCD unit waits in the done state until cycle 310 when it is able to send the result to the test sink. On cycle 311 the GCD unit accepts a new input message to work on. This is a great example of how an effective line trace can enable you to quickly visualize how a design is actually working.

```
cycle src        A    B    Areg Breg ST out    sink

 296: 002d00e1 > 00e1:002d(0002 0000 I )     >
 297: #        > #        (00e1 002d C-)     >
 298: #        > #        (00b4 002d C-)     >
 299: #        > #        (0087 002d C-)     >
 300: #        > #        (005a 002d C-)     >
 301: #        > #        (002d 002d C-).    > .
 302: #        > #        (0000 002d Cs)     >
 303: #        > #        (002d 0000 C )     >
 304: #        > #        (002d 0000 D )#    > #
 305: #        > #        (002d 0000 D )#    > #
 306: #        > #        (002d 0000 D )#    > #
 307: #        > #        (002d 0000 D )#    > #
 308: #        > #        (002d 0000 D )#    > #
 309: #        > #        (002d 0000 D )#    > #
 310: #        > #        (002d 0000 D )002d > 002d
 311: 002200cc > 00cc:0022(002d 0000 I )     >
 312: #        > #        (00cc 0022 C-)     >
 313: #        > #        (00aa 0022 C-)     >
 314: #        > #        (0088 0022 C-)     >
```

Figure 58: 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.

★ *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.

6.3. Evaluating the GCD Unit using a Simulator

As with the previous section, we have provided a simulator for evaluating the performance of the GCD implementation. In this case, we are focusing on a single implementation with two different input datasets. You can run the simulator and look at the average number of cycles to compute a GCD for each input dataset on the FL model like this:

```
% cd ${TUTROOT}/build
% ../tut3_verilog/gcd/gcd-sim --stats --impl fl --input small
% ../tut3_verilog/gcd/gcd-sim --stats --impl fl --input random
```

Not surprisingly since the FL model is just a functional-level model with no timing information, the number of cycles per GCD transaction is always about one. You can run the simulator and look at the average number of cycles to compute a GCD for each input dataset on the RTL model like this:

```
% cd ${TUTROOT}/build
% ../tut3_verilog/gcd/gcd-sim --stats --impl rtl --input small
% ../tut3_verilog/gcd/gcd-sim --stats --impl rtl --input random
```

Here we can see that on average it takes fewer cycles per GCD transaction when operating on small integers versus large random integers. You can generate and view a waveform for the simulation like this:

```
% cd ${TUTROOT}/build
% ../tut3_verilog/gcd/gcd-sim --impl rtl --input random --dump-vcd
% open gcd-rtl-random_top_gcd.verilator1.vcd
```

Acknowledgments

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

Appendix A: Constructs Allowed in Synthesizable RTL

Always Allowed in Synthesizable RTL	**Allowed in Synthesizable RTL With Limitations**	**Explicitly Not Allowed in Synthesizable RTL**
`logic`	`always`[1]	`wire, reg`[15]
`logic [N-1:0]`	`enum`[2]	`integer, real, time, realtime`
`& | ^ ^~ ~` (bitwise)	`struct`[3]	`signed`[16]
`&& || !`	`casez, endcase`[4]	`===, !==`
`& ~& | ~| ^ ^~` (reduction)	`task, endtask`[5]	`* / % **`
`+ -`	`function, endfunction`[5]	`#N` (delay statements)
`>> << >>>`	`=` (blocking assignment)[6]	`inout`[17]
`== != > <= < <=`	`<=` (non-blocking assignment)[7]	`initial`
`{}`	`typedef`[8]	variable initialization[18]
`{N{}}` (repeat)	`packed`[9]	`negedge`[19]
`?:`	`$clog2()`[10]	`casex, endcase`
`always_ff, always_comb`	`$bits()`[10]	`for, while, repeat, forever`[20]
`if else`	`$signed()`[11]	`fork, join`
`case, endcase`	read-modify-write `signal`[12]	`deassign, force, release`
`begin, end`	`*`[13]	`specify, endspecify`
`module, endmodule`	`for`[14]	`nmos, pmos, cmos`
`input, output`		`rnmos, rpmos, rcmos`
`assign`		`tran, tranif0, tranif1`
`parameter`		`rtran, rtranif0, rtranif1`
`localparam`		`supply0, supply1`
`genvar`		`strong0, strong1`
`generate, endgenerate`		`weak0, weak1`
`generate for`		`primitive, endprimitive`
`generate if else`		`defparam`
`generate case`		unnamed port connections[21]
named port connections		unnamed parameter passing[22]
named parameter passing		all other keywords
		all other system tasks

1 Students should prefer using `always_ff` and `always_comb` instead of `always`. If students insist on using `always`, then it can only be used in one of the following two constructs: `always @(posedge clk)` for sequential logic, and `always @(*)` for combinational logic. Students are not allowed to trigger sequential blocks off of the negative edge of the clock or create asynchronous resets, nor use explicit sensitivity lists.

2 `enum` can only be used with an explicit base type of `logic` and explicitly setting the bitwidth using the following syntax: `typedef enum logic [$clog2(N)-1:0] { ... } type_t;` where `N` is the number of labels in the `enum`. Anonymous `enums` are not allowed.

3 `struct` can only be used with the `packed` qualifier (i.e., `unpacked` structs are not allowed) using the following syntax: `typedef struct packed { ... } type_t;` Anonymous `structs` are not allowed.

4 `casez` can only be used in very specific situations to compactly implement priority encoder style hardware structures.

5 `task` and `function` blocks must themselves contain only synthesizable RTL.

6 Blocking assignments should only be used in `always_comb` blocks and are explicitly not allowed in `always_ff` blocks.

7 Non-blocking assignments should only be used in `always_ff` blocks and are explicitly not allowed in `always_comb` blocks.

8 `typedef` should only be used in conjunction with `enum` and `struct`.

9 `packed` should only be used in conjunction with `struct`.

10 The input to `$clog2/$bits` must be a static-elaboration-time constant. The input to `$clog2/$bits` cannot be a signal (i.e., a wire or a port). In other words, `$clog2/$bits` can only be used for static elaboration and cannot be used to model actual hardware.

11 `$signed()` can only be used around the operands to `>>>`, `>`, `>=`, `<`, `<=` to ensure that these operators perform the signed equivalents.

12 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 `always_comb` concurrent block. This is a combinational loop and does not model valid hardware. Read-modify-write is allowed in an `always_ff` concurrent block with a non-blocking assignment, although we urge students to consider separating the sequential and combinational logic. Students can use an `always_comb` concurrent block to read the signal, perform some arithmetic on the corresponding value, and then write a temporary wire; and use an `always_ff` concurrent block to flop the temporary wire into the destination signal.

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

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

15 `wire` and `reg` are perfectly valid, synthesizable constructs, but `logic` is much cleaner. So we would like students to avoid using `wire` and `reg`.

16 `signed` types can sometimes be synthesized, but we do not allow this construct in the course.

17 Ports with `inout` can be used to create tri-state buses, but tools often have trouble synthesizing hardware from these kinds of models.

18 Variable initialization means assigning an initial value to a `logic` variable when you declare the variable. This is not synthesizable; it is not modeling real hardware. If you need to set some state to an initial condition, you must explicitly use the `reset` signal.

19 Triggering a sequential block off of the `negedge` of a signal is certainly synthesizable, but we will be exclusively using a positive-edge-triggered flip-flop-based design style.

20 If you would like to generate hardware using loops, then you should use `generate` blocks.

21 In very specific, rare cases unnamed port connections might make sense, usually when there are just one or two ports and there purpose is obvious from the context.

22 In very specific, rare cases unnamed parameter passing might make sense, usually when there are just one or two parameters and their purpose is obvious from the context.