ECE 5745 Complex Digital ASIC Design

Tutorial 4: Verilog Hardware Description Language

School of Electrical and Computer Engineering
Cornell University

revision: 2022-03-31-18-08

Contents

1. Introduction

In the lab assignments for this course, we will be using the PyMTL3 hardware modeling framework for functional-level modeling, verification, and simulator harnesses. Students can choose to use either PyMTL3 or Verilog to do their register-transfer-level (RTL) modeling. If you are planning to use PyMTL3, then you do not need to complete this tutorial. If you are planning to use Verilog, you should still complete or at least skim the PyMTL3 tutorial since we will always be using PyMTL3 for some aspects of the lab assignment.

Please note that this tutorial describes using PyMTL3 which is a much improved new version compared to PyMTL2. You may have used PyMTL2 in ECE 4750. For the post part, the differences in terms of the API between PyMTL2 and PyMTL3 are modest. Most of the changes are in the actual implementation of this API. However, there definitely are changes that you will need to be aware of.

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. Although we will be using the open-source tool Icarus Verilog (`iverilog`) for compiling some simple Verilog models into simulators, we will primarily be using the open-source tool Verilator (`verilator`) as our Verilog simulator. PyMTL3 has built-in support for testing Verilog simulators created using Verilator. As in the PyMTL3 tutorial, we will also be using GTKWave (`gtkwave`) for viewing `.vcd` waveforms. 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: (1) Windows PowerShell or Mac OS X Terminal; (2) VS Code; or (3) X2Go. 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-ece5745.sh script** or that you have added it to your `.bashrc` script, which will then source the script every time you login. Sourcing the setup script sets up the environment required for this 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-ece5745/ece5745-tut4-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/ece5745-tut4-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-ece5745.sh
% mkdir -p ${HOME}/ece5745
% cd ${HOME}/ece5745
% git clone https://github.com/githubid/ece5745-tut4-verilog.git tut4
% cd tut4/sim
% TUTROOT=${PWD}
```

> **NOTE:** It should be possible to experiment with this tutorial even if you are not enrolled in the course and/or do not have access to the course computing resources. All of the code for the tutorial is located on GitHub. You will not use the `setup-ece5745.sh` script, and your specific environment may be different from what is assumed in this tutorial.

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 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 the student includes non-synthesizable code in their actual design (i.e., not the test harness), 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.

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

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
```

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).

```
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.

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.

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.

```
1  module top;
2
3    // Declare single-bit logic variables.
4
5    logic a;
6    logic b;
7    logic c;
8
9    initial begin
10
11     // Single-bit literals
12
13     a = 1'b0;    $display( "1'b0   = %x ", a );
14     a = 1'b1;    $display( "1'b1   = %x ", a );
15     a = 1'bx;    $display( "1'bx   = %x ", a );
16     a = 1'bz;    $display( "1'bz   = %x ", a );
17
18     // Bitwise logical operators for doing AND, OR, XOR, and NOT
19
20     a = 1'b0;
21     b = 1'b1;
22
23     c = a & b;   $display( "0 & 1  = %x ", c );
24     c = a | b;   $display( "0 | 1  = %x ", c );
25     c = a ^ b;   $display( "0 ^ 1  = %x ", c );
26     c = ~b;      $display( "~1     = %x ", c );
27
28     // Bitwise logical operators for doing AND, OR, XOR, and NOT with X
29
30     a = 1'b0;
31     b = 1'bx;
32
33     c = a & b;   $display( "0 & x  = %x ", c );
34     c = a | b;   $display( "0 | x  = %x ", c );
35     c = a ^ b;   $display( "0 ^ x  = %x ", c );
36     c = ~b;      $display( "~x     = %x ", c );
37
38     // Boolean logical operators
39
40     a = 1'b0;
41     b = 1'b1;
42
43     c = a && b;  $display( "0 && 1 = %x ", c );
44     c = a || b;  $display( "0 || 1 = %x ", c );
45     c = !b;      $display( "!1     = %x ", c );
46
47   end
48
49 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.

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.

★ *To-Do On Your Own:* Experiment different multi-bit logic values and shift amounts.

```
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 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.

```
module top;

  // Declare multi-bit logic variables

  logic        a; // 1-bit logic variable
  logic [ 3:0] A; // 4-bit logic variable
  logic [ 3:0] B; // 4-bit logic variable

  initial begin

    // Relational operators

    A = 4'd15; B = 4'd09;

    a = ( A == B );   $display( "(15 == 9) = %x", a );
    a = ( A != B );   $display( "(15 != 9) = %x", a );
    a = ( A >  B );   $display( "(15 >  9) = %x", a );
    a = ( A >= B );   $display( "(15 >= 9) = %x", a );
    a = ( A <  B );   $display( "(15 <  9) = %x", a );
    a = ( A <= B );   $display( "(15 <= 9) = %x", a );

    // Relational operators when some bits are X

    A = 4'b1100; B = 4'b10xx;

    a = ( A == B );   $display( "(4'b1100 == 4'b10xx) = %x", a );
    a = ( A != B );   $display( "(4'b1100 != 4'b10xx) = %x", a );
    a = ( A >  B );   $display( "(4'b1100 >  4'b10xx) = %x", a );
    a = ( A <  B );   $display( "(4'b1100 <  4'b10xx) = %x", a );

    // Signed relational operators

    A = 4'b1111; // -1 in twos complement
    B = 4'd0001; //  1 in twos complement

    a = (         A  >         B  ); $display( "(-1 > 1) = %x", a );
    a = (         A  <         B  ); $display( "(-1 < 1) = %x", a );
    a = ( $signed(A) > $signed(B) ); $display( "(-1 > 1) = %x", a );
    a = ( $signed(A) < $signed(B) ); $display( "(-1 < 1) = %x", a );

  end

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.

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

```
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; // 18-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 waveforms, ad-hoc testing, and a simple unit testing framework. As with PyMTL3 RTL design, 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.

4.1. RTL Model of Registered Incrementer

Figure 16 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 16. 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 16 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 Python, Verilog does not have a clean way to manage namespaces for macros and module names. This means that if 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 `tut4_verilog/regincr`, we will use `TUT4_VERILOG_REGINCR_` as a prefix for all macro names and `tut4_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 always, 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 16 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 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`).

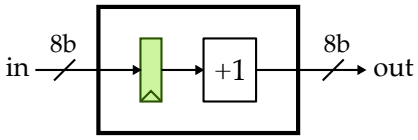

Figure 14: Block Diagram for Registered Incrementer – An eight-bit registered incrementer with an eight-bit input port, an eight-bit output port, and an (implicit) clock input.

```
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 TUT4_VERILOG_REGINCR_REG_INCR_V
12 `define TUT4_VERILOG_REGINCR_REG_INCR_V
13
14 module tut4_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 /* TUT4_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 TUT4_VERILOG_REGINCR_REG_INCR_V
12 `define TUT4_VERILOG_REGINCR_REG_INCR_V
13
14 module tut4_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 /* TUT4_VERILOG_REGINCR_REG_INCR_V */
```

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

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 (analogous to PyMTL3 `@s.tick_rtl` concurrent blocks) and `always_comb` concurrent blocks to model combinational logic (analogous to PyMTL3 `@s.combinational` concurrent blocks). 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.

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 `tut4_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. Simulating a Model using `iverilog`

Now that we have developed a new hardware module, we can test its functionality using a simulation harness. Figure 17 shows an ad-hoc test 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

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

`include "../tut4_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;

  tut4_verilog_regincr_RegIncr reg_incr
  (
    .clk   (clk),
    .reset (reset),
    .in_   (in_),
    .out   (out)
  );

  // Verify functionality

  initial begin

    // Dump waveforms

    $dumpfile("regincr-iverilog-sim.vcd");
    $dumpvars;

    // Reset

    #11;
    reset = 1'b0;

    // Test cases

    in_ = 8'h00;
    #10;
    if ( out != 8'h01 ) begin
      $display( "ERROR: out, expected = %x, actual = %x", 8'h01, out );
      $finish;
    end

    in_ = 8'h13;
    #10;
    if ( out != 8'h14 ) begin
      $display( "ERROR: out, expected = %x, actual = %x", 8'h14, out );
      $finish;
    end

    in_ = 8'h27;
    #10;
    if ( out != 8'h28 ) begin
      $display( "ERROR: out, expected = %x, actual = %x", 8'h28, out );
      $finish;
    end

    $display( "*** PASSED ***" );
    $finish;
  end

endmodule
```

Figure 17: Simulator for Registered Incrementer – A Verilog simulator for the eight-bit registered incrementer in Figure 16.

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 19–24 instantiate the device 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 21–23 illustrate the correct coding convention with one port binding per line and the ports/nets vertically aligned. As shown on lines 20 and 24 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 28–62 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 32–33 instruct the simulator to dump waveforms for all nets. Line 35 is a delay statement that essentially waits for one timestep. Delay statements are not synthesizable and should only be used in test harnesses. Lines 39–44 implement a test by setting the inputs of the device under test, waiting for 10 time steps, and then checking that the output is as expected. If there is an error, we display an error message and stop the simulation. We include two more tests, and if we make it to the bottom of the `initial` block then the test has passed.

Edit the Verilog simulation harness named `regincr-iverilog-sim.v` in the `tut4_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}/build
% iverilog -g2012 -o regincr-iverilog-sim ../tut4_verilog/regincr/regincr-iverilog-sim.v
% ./regincr-iverilog-sim
```

If everything goes as expected, then the ad-hoc test should display `*** PASSED ***`.

★ *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 tests no longer pass. Modify the ad-hoc test so that it includes the correct reference outputs for a +2 incrementer, recompile, rerun the ad-hoc test, and verify that the test now pass. When you are finished, edit the register incrementer so that it again increments by +1.

4.3. Verifying a Model with Unit Testing

Writing test and simulation 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 PyMTL3 for FL modeling, but also to productively write test and simulation harnesses for our Verilog RTL models. PyMTL3 includes support for *Verilog import* by writing a special PyMTL3 wrapper model. Once we have

```
1  #=========================================================================
2  # RegIncr
3  #=========================================================================
4  # This is a simple model for a registered incrementer. An eight-bit value
5  # is read from the input port, registered, incremented by one, and
6  # finally written to the output port.
7
8  from pymtl3 import *
9  from pymtl3.passes.backends.verilog import *
10
11 class RegIncr( VerilogPlaceholder, Component ):
12
13   # Constructor
14
15   def construct( s ):
16
17     # Port-based interface
18
19     s.in_ = InPort ( 8 )
20     s.out = OutPort( 8 )
21
22     # The port map by default uses the PyMTL3 port names
23     # s.set_metadata( VerilogPlaceholderPass.port_map, {
24     #   s.in_: 'in_',
25     #   s.out: 'out',
26     # })
27
28     # has_clk and has_reset are True by default
29     # s.set_metadata( VerilogPlaceholderPass.has_clk, True )
30     # s.set_metadata( VerilogPlaceholderPass.has_reset, True )
```

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

created this wrapper model, we can use all of the sophisticated techniques we learned in the PyMTL3 tutorial for writing test and simulation 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 `tut4_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 `tut4_verilog_regincr_RegIncr` as the Verilog model name. The interface on lines 19–20 should be identical to what we would use with a standard PyMTL3 implementation. We will 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.

Once we have a PyMTL3 wrapper, we can use the exact same test harness we used in the PyMTL3 tutorial. The following commands will run the `RegIncr_test.py` test script.

```
% cd ${TUTROOT}/build
% pytest ../tut4_verilog/regincr/test/RegIncr_test.py -sv
% pytest ../tut4_verilog/regincr/test/RegIncr_extra_test.py -sv
```

Keep in mind that PyMTL3 uses the open-source `verilator` simulator instead of `iverilog` for simulating Verilog modules. The framework will output the specific `verilog` command being used to create the simulator. You can us all of the `pytest` features we learned in the PyMTL3 tutorial including using the `-s` option to output line tracing and the `--dump-vcd` option to output VCD.

```
% cd ${TUTROOT}/build
% pytest ../tut4_verilog/regincr/test/RegIncr_test.py -s --dump-vcd
% gtkwave tut4_verilog.regincr.test.RegIncr_test__test_basic_top.verilator1.vcd
```

Note that the actual name of the VCD file will be a little different compared to when we use PyMTL3 RTL models. Also note that PyMTL3 will actually create *two* different VCD files. You always want to open the one with the `verilator1.vcd` suffix. In the lab assignments, we will mostly provide you with the appropriate top-level PyMTL3 wrappers, but you may need to write your own wrappers for new child models you want to test.

As we learned in the PyMTL3 tutorial, when testing an entire directory, we can use an iterative process to "zoom" in on a failing test case. We will 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 will 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 ../tut4_verilog/regincr
% pytest ../tut4_verilog/regincr/test/RegIncr2stage_test.py -v
% pytest ../tut4_verilog/regincr/test/RegIncr2stage_test.py -v -x --tb=short
```

★ *To-Do On Your Own:* Edit the register incrementer so that it now increments by +2 instead of +1. Recompile, rerun the unit test, and verify that the tests no longer pass. Modify the unit test so that it includes the correct reference outputs for a +2 incrementer, recompile, rerun the unit test, and verify that the test now pass. When you are finished, edit the register incrementer so that it again increments by +1.

4.4. Reusing a Model with Structural Composition

As in PyMTL3, we can use modularity and hierarchy to structurally compose small, simple models into large, complex models. Figure 19 shows a two-stage registered incrementer that uses structural composition to instantiate and connect two instances of a single-stage registered incrementer. Figure 20 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.

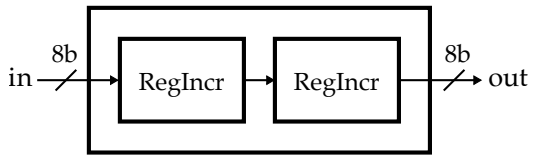

Figure 19: Block Diagram for Two-Stage Registered Incrementer – An eight-bit two-stage registered incrementer that reuses the registered incrementer in Figure 14 through structural composition.

```
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 TUT4_VERILOG_REGINCR_REG_INCR_2STAGE_V
9  `define TUT4_VERILOG_REGINCR_REG_INCR_2STAGE_V
10
11 `include "tut4_verilog/regincr/RegIncr.v"
12
13 module tut4_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   tut4_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   tut4_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 /* TUT4_VERILOG_REGINCR_REG_INCR_2STAGE_V */
```

Figure 20: Two-Stage Registered Incrementer – An eight-bit two-stage registered incrementer corresponding to Figure 19. This model is implemented using structural composition to instantiate and connect two instances of the single-stage register incrementer.

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.

Edit the Verilog source file named `RegIncr2stage.v`. Add lines 33-41 from Figure 20 to instantiate and connect the second stage of the two-stage registered incrementer. Then reuse the test harness from the PyMTL3 tutorial as follows:

```
% cd ${TUTROOT}/build
% pytest ../tut4_verilog/regincr/test/RegIncr2stage_test.py -sv
```

★ *To-Do On Your Own:* Create a three-stage registered incrementer similar in spirit to the two-stage registered incrementer in Figure 19. Verify your design by writing a test script that uses test vectors.

4.5. Parameterizing a Model with Static Elaboration

As we learned in the PyMTL3 tutorial, 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 for only eight-bit input values, but we may want to reuse this model in a different context with four-bit input values or 16-bit input values. We can use Verilog *parameters* to parameterize the port bitwidth for the registered incrementer shown in Figure 16; we would replace references to 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 21 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.

Unfortunately, writing highly parameterized models in Verilog can be very tedious or even impossible, which is one key motivation for the PyMTL3 framework. Having said this, Verilog-2001 does provide `generate` statements which are meant for static elaboration. Recall that static elaboration happens at compile time, not runtime. We can use static elaboration to *generate* hardware which is fundamentally different from *modeling* hardware. Figure 22 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_nstage` parameter, we cannot directly reuse the previous wrapper for `RegIncr` which does not have parameters. Figure 23 shows how to map the `nstages` parameter of the PyMTL3 component to `p_nstages` in the Verilog file using `s.set_metadata` API on lines 24–26 where we set the `VerilogPlaceholderPass.params` variable to a parameter dictionary. Note that if the Verilog model also has `nstages` as parameter, you don't need to explicitly set the metadata.

We can reuse the test harness from the PyMTL3 tutorial as follows:

```
% cd ${TUTROOT}/build
% pytest ../tut4_verilog/regincr/test/RegIncrNstage_test.py -sv
```

```
module vc_Incrementer
#(
  parameter p_nbits     = 1,
  parameter p_inc_value = 1
)(
  input  logic [p_nbits-1:0] in,
  output logic [p_nbits-1:0] out
);

  assign out = in + p_inc_value;

endmodule
```

Figure 21: 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 TUT4_VERILOG_REGINCR_REG_INCR_NSTAGE_V
`define TUT4_VERILOG_REGINCR_REG_INCR_NSTAGE_V

`include "tut4_verilog/regincr/RegIncr.v"

module tut4_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

    tut4_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 /* TUT4_VERILOG_REGINCR_REG_INCR_NSTAGE_V */
```

Figure 22: 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
11   # Constructor
12
13   def construct( s, nstages=2 ):
14
15     # Port-based interface
16
17     s.in_ = InPort ( 8 )
18     s.out = OutPort( 8 )
19
20     # If the Verilog parameter name is the same as PyMTL3 component
21     # (e.g., here the Verilog source takes 'nstages' instead of 'p_nstages'),
22     # we don't need to set this metadata.
23
24     s.set_metadata( VerilogPlaceholderPass.params, {
25       'p_nstages' : nstages,
26     })
```

Figure 23: Registered Incrementer Wrapper Parameterized by the Number of Stages – PyMTL3 wrapper for the `RegIncrNstage` Verilog module.

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 this section, we will explore how we can implement the same sorting unit previously seen in the PyMTL3 tutorial except using the Verilog hardware description language. As a reminder, the simple pipelined four-element sorter is shown in Figure 24. 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. Most of the code for this section is provided for you in the `tut4_verilog/sort` subdirectory.

5.1. Flat Sorter Implementation

We will begin by exploring a flat implementation of the sorter that does not instantiate any additional child modules. This implementation is provided for you in the file named `SortUnitFlatRTL.v`. Figure 25 illustrates the Verilog code that describes the interface for the sorter. 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.

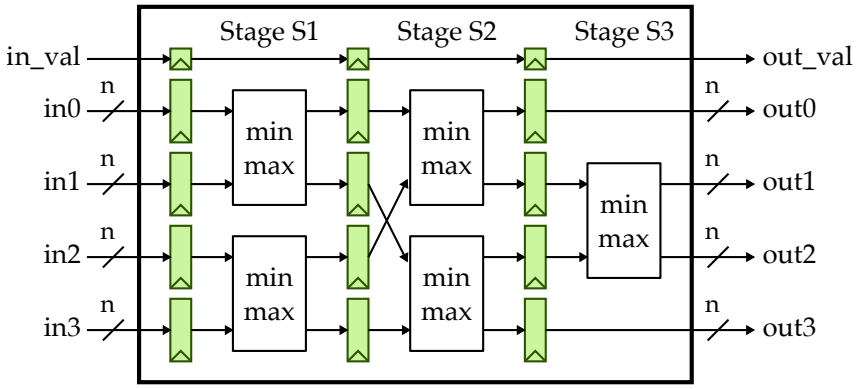

Figure 24: Block Diagram for Four-Element Sorter – An n-bit pipelined four-element sorter which arranges the four elements in ascending order.

```
1  module tut4_verilog_sort_SorterFlat
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 25: Interface for the Four-Element Sorter – The interface corresponds to the diagram in Figure 24 and is parameterized by the bitwidth of each element.

Figure 26 shows the first pipeline stage of the flat implementation of the sorter. Notice how we use the parameter `p_nbits` to declare various internal registers and wires. We cleanly separate the sequential logic from the combinational logic. We use comments and explicit suffixes to make it clear what pipeline stage we are modeling.

The corresponding wrapper is in `SortUnitFlatRTL.py` and is shown in Figure 27. On lines 24–26, we specify how the PyMTL3 parameters (e.g., `nbits`) correspond to the Verilog parameters (e.g., `p_nbits`). Here, we need to set the port map, because we are using an array of ports in PyMTL3, but the Verilog model has eight different ports. Note that the `VerilogImportPass` will scan for `VC_TRACE_BEGIN` to check whether the Verilog module includes its own line tracing code, and include it in the simulation. We can run the unit tests for this module as follows:

```
% cd ${TUTROOT}/build
% pytest ../tut4_verilog/sort/test/SortUnitFlatRTL_test.py
```

The design should pass all of the tests.

```
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 26: First Stage of the Flat Sorter Implementation – First pipeline stage of the sorter using a flat implementation corresponding to the diagram in Figure 24.

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

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

class SortUnitFlatRTL( VerilogPlaceholder, Component ):

  # Constructor

  def construct( s, nbits=8 ):

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

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

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

    s.set_metadata( VerilogPlaceholderPass.params, {
      'p_nbits' : nbits
    })

    s.set_metadata( VerilogPlaceholderPass.port_map, {
      s.in_[0]  : 'in0',
      s.in_[1]  : 'in1',
      s.in_[2]  : 'in2',
      s.in_[3]  : 'in3',
      s.out[0]  : 'out0',
      s.out[1]  : 'out1',
      s.out[2]  : 'out2',
      s.out[3]  : 'out3',
    })
```

Figure 27: Sort Unit Wrapper – PyMTL3 wrapper for the Verilog RTL implementation of the flat sort unit.

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

5.2. Using Verilog Line Traces

We learned about line tracing in the PyMTL3 tutorial. We have implemented a small library that enables implementing line tracing code directly within your Verilog modules. While not as elegant as PyMTL3 line tracing, Verilog line tracing will still be an essentially way to visualize our designs. You can use the the `-s` command-line option to see a line trace of the four-element sorter.

```
% cd ${TUTROOT}/build
% pytest ../tut4_verilog/sort/test/SortUnitFlatRTL_test.py -k test_basic -s
```

Figure 28 shows a portion of a representative line trace with some additional annotation. The first column indicates the current cycle. There are fixed-width columns showing the inputs and outputs of the module along with the state in the S1, S2, and S3 pipeline registers at the beginning of the cycle. Notice that we use spaces when the data is invalid which improves readability and makes it easy to see when the hardware is actually doing useful work. If you compare this line trace to the line trace generated in the PyMTL3 tutorial, they should look identical. Both PyMTL3 and Verilog enable the same kind of cycle-accurate modeling. Regardless of whether we are using PyMTL3 or Verilog, line traces are a powerful way to visualize your design and debug both correctness and performance issues.

```
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 28: 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.

```
1  `ifndef SYNTHESIS
2
3  logic [`VC_TRACE_NBITS-1:0] str;
4  `VC_TRACE_BEGIN
5  begin
6
7    // Inputs
8
9    $sformat( str, "{%x,%x,%x,%x}", in0, in1, in2, in3 );
10   vc_trace.append_val_str( trace_str, in_val, str  );
11   vc_trace.append_str( trace_str, "|" );
12
13   ...
14
15 end
16 `VC_TRACE_END
17
18 `endif /* SYNTHESIS */
```

Figure 29: Example of Line Tracing Code – This code generates the first fixed-width column for the line trace shown in Figure 28

As mentioned above, we provide a verilog component (VC) library with various useful Verilog modules, including one module in `vc/trace.v` which makes it easier to create line traces in your own Verilog modules. Figure 29 shows a small snippet of code that is used in the sorter implementation to trace the input ports. Notice how we have wrapped the line tracing code in `` `ifndef SYNTHESIS `` and `` `endif `` to clearly indicate that this code is not synthesizable even though it is included in out design. Line 3 declares a temporary string variable that we will use when converting nets into strings. Lines 4–16 use helper macros to declare a task called `trace` which takes a special argument called `trace_str` that holds the current line trace string. The job of this task is to append trace information to the `trace_str` that describes the current state and operation of this module. The `vc_trace.append_str` and `vc_trace.append_val_str` helper tasks add strings to the line trace. You can also build line traces hierarchically by explicitly calling the `trace` task on a child module. Although we will provide significant line tracing code in each lab harness, you are also strongly encouraged to augment this code with your own line tracing.

★ *To-Do On Your Own:* Modify the line tracing code to show the pipeline stage label (`in_`, `S1`, `S2`, `S3`, `out`) before each stage. After your modifications, the line trace might look something like this:

```
24: in:{05,07,06,08}|S1:{a5,a3,a2,a7}|S2:{03,04,01,02}|S3:             |out:
```

5.3. Structural Sorter Implementation

The flat implementation in `SortUnitFlatRTL.v` is complex and monolithic and it fails to really exploit the structure inherent in the sorter. Just as in the PyMTL3 tutorial, 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. We have started a structural implementation in `SortUnitStructRTL.v`; the structural implementation will have an identical interface and behavior as the flat implementation in `SortUnitFlatRTL.v`.

Figure 30 shows the first pipeline stage of the structural implementation of the sorter. Our design instantiates three kinds of modules: `vc_ResetReg`, `vc_Reg`, and `tut4_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 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.

```
//------------------------------------------------------------------------
// 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;

tut4_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;

tut4_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 30: First Stage of the Structural Sorter Implementation – First pipeline stage of the sorter using a structural implementation corresponding to the diagram in Figure 24.

★ *To-Do On Your Own:* 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. Once your min/max unit is complete and tested, then test the structural sorter implementation like this:

```
% cd ${TUTROOT}/build
% pytest ../tut4_verilog/sort/test/SortUnitStructRTL_test.py -v
% pytest ../tut4_verilog/sort/test/SortUnitStructRTL_test.py -k test_basic -s
```

The line trace for the sort unit structural RTL model should be the same as in Figure 28, since these are really just two different implementations of the sort unit RTL.

5.4. Evaluating the Sorter Using a Simulator

Just like we can use PyMTL3 test harnesses to verify Verilog modules, we can also use PyMTL3 simulation harnesses for evaluation. Since our PyMTL3 wrapper has the exact same interface as a PyMTL3 RTL implementation, the exact same simulator from the PyMTL3 tutorial will work without change.

```
% cd ${TUTROOT}/build
% ../tut4_verilog/sort/sort-sim --stats --impl rtl-flat
% ../tut4_verilog/sort/sort-sim --stats --impl rtl-struct
```

6. Greatest Common Divisor: Verilog Design Example

In this section, we will apply what we have learned in the previous section to study a more complicated hardware unit that calculates the greatest common divisor (GCD) of two input operands. Our design will be exactly the same as what we experimented with in the PyMTL3 tutorial, except now we will be using the Verilog hardware description language. The code for this section is provided for you in the `tut4_verilog/gcd` subdirectory. The previous examples placed the unit test scripts in the same subdirectory as the models these tests were testing. As we start to explore much larger and more complicated designs, it can be useful to keep all of the unit tests together in a separate `test` subdirectory. You can see in this example, that all of the unit tests for the GCD unit are placed in the `tut4_verilog/gcd/test` subdirectory.

As a reminder, Figure 31 illustrates the interface for our module. The GCD unit will take two n-bits operands and produce an n-bit result. For flow-control we use the same latency-insensitive stream val/rdy interface that we learned about in the PyMTL3 tutorial. Our implementation will use Euclid's algorithm to iteratively calculate the GCD. Figure 32 illustrates this algorithm as an executable Python function.

Figure 33 shows that we still don't need to set additional metadata even if we use stream interfaces, as long as the port names in the Verilog module is mangled in our convention. For example, `s.recv.rdy` corresponds to `recv_rdy` port of the GCD Verilog interface.

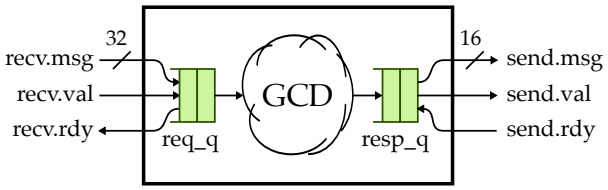

Figure 31: Functional-Level Implementation of GCD Unit – Input and output use latency-insensitive val/rdy interfaces. The input message includes two 16-bit operands; output message is an 16-bit result. Clock and reset signals are not shown.

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

Figure 32: 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.

```
1  #=========================================================================
2  # GCD Unit RTL Model
3  #=========================================================================
4
5  from pymtl3 import *
6  from pymtl3.stdlib import stream
7  from pymtl3.passes.backends.verilog import *
8
9  from .GcdUnitMsg  import GcdUnitMsgs
10
11 class GcdUnitRTL( VerilogPlaceholder, Component ):
12
13   # Constructor
14
15   def construct( s ):
16
17     # Interface
18
19     s.recv = stream.ifcs.RecvIfcRTL( GcdUnitMsgs.req )
20     s.send = stream.ifcs.SendIfcRTL( GcdUnitMsgs.resp )
```

Figure 33: GCD Unit Wrapper – PyMTL3 wrapper for the Verilog RTL implementation of the gcd unit.

★ *To-Do On Your Own:* Experiment with the algorithm using a Python interpreter. Try calculating the GCD for different input values. Add additional debugging output to track what the algorithm is doing each iteration.

6.1. Control/Datapath Split Implementation

As discussed in the PyMTL3 tutorial, we will usually divide more complicated designs 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 34 illustrates the datapath for the GCD unit and Figure 35 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 `GcdUnitRTL.v`.

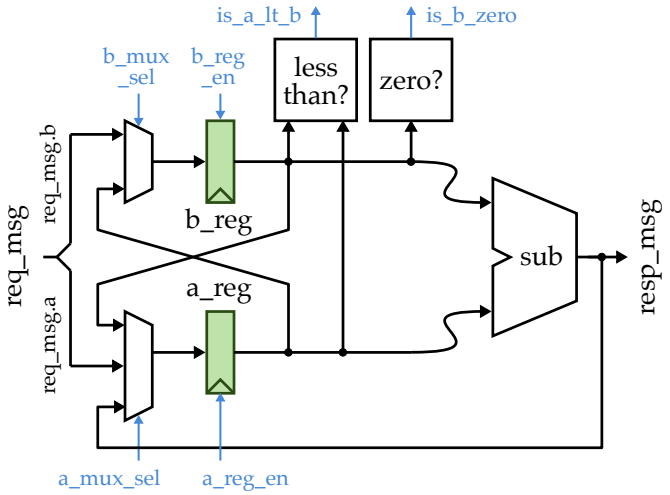

Figure 34: Datapath Diagram for GCD – Datapath includes two state registers and required muxing and arithmetic units to iteratively implement Euclid's algorithm.

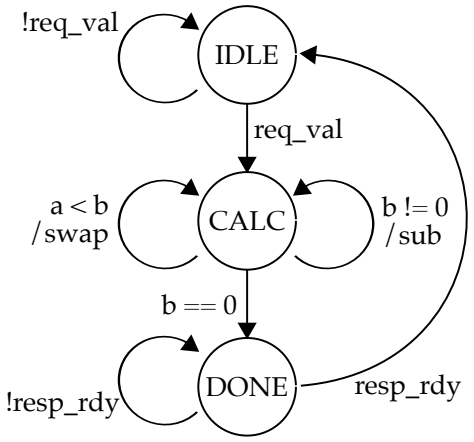

Figure 35: FSM Diagram for GCD – A hybrid Moore/Mealy FSM for controlling the datapath in Figure 34. Mealy transitions in the calc state determine whether to swap or subtract.

Take a look at the datapath interface which is also shown in Figure 36. We cleanly identify the data signals, control signals, and status signals. Figure 36 also shows the first two datapath components, but take a look in `GcdUnitRTL.v` to see the entire datapath. Notice how we use a very structural implementation that *exactly* matches the datapath diagram in Figure 34. 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 use case statements to compactly represent the state transitions and outputs. Figure 37 shows the portion of the FSM responsible for setting the output signals. We use a task to set all of the control signals in a single line; as long as the task does not include non-synthesizable constructs (e.g., delay statements or system tasks) the task itself should 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 38 shows a portion of the top-level module that connects the datapath and control unit together. Lines X and Y 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.

```
1  module tut4_verilog_gcd_GcdUnitDpath
2  (
3    input  logic        clk,
4    input  logic        reset,
5
6    // Data signals
7
8    input  logic [31:0] recv_msg,
9    output logic [15:0] send_msg,
10
11   // Control signals
12
13   input  logic        a_reg_en,   // Enable for A register
14   input  logic        b_reg_en,   // Enable for B register
15   input  logic [1:0]  a_mux_sel,  // Sel for mux in front of A reg
16   input  logic        b_mux_sel,  // sel for mux in front of B reg
17
18   // Status signals
19
20   output logic        is_b_zero,  // Output of zero comparator
21   output logic        is_a_lt_b   // Output of less-than comparator
22 );
23
24   localparam c_nbits = 16;
25
26   // Split out the a and b operands
27
28   logic [c_nbits-1:0] req_msg_a = recv_msg[31:16];
29   logic [c_nbits-1:0] req_msg_b = recv_msg[15:0 ];
30
31   // A Mux
32
33   logic [c_nbits-1:0] b_reg_out;
34   logic [c_nbits-1:0] sub_out;
35   logic [c_nbits-1:0] a_mux_out;
36
37   vc_Mux3#(c_nbits) a_mux
38   (
39     .sel   (a_mux_sel),
40     .in0   (req_msg_a),
41     .in1   (b_reg_out),
42     .in2   (sub_out),
43     .out   (a_mux_out)
44   );
45
46   // A register
47
48   logic [c_nbits-1:0] a_reg_out;
49
50   vc_EnReg#(c_nbits) a_reg
51   (
52     .clk   (clk),
53     .reset (reset),
54     .en    (a_reg_en),
55     .d     (a_mux_out),
56     .q     (a_reg_out)
57   );
```

Figure 36: 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 task cs
15 (
16   input logic       cs_recv_rdy,
17   input logic       cs_send_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   recv_rdy  = cs_recv_rdy;
25   send_val  = cs_send_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 endtask
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 37: 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 tut4_verilog_gcd_GcdUnitRTL
2  (
3    input  logic             clk,
4    input  logic             reset,
5
6    input  logic             recv_val,
7    output logic             recv_rdy,
8    input  logic [31:0]      recv_msg,
9
10   output logic             send_val,
11   input  logic             send_rdy,
12   output logic [15:0]      send_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   tut4_verilog_gcd_GcdUnitCtrl ctrl
34   (
35     .*
36   );
37
38   // Datapath
39
40   tut4_verilog_gcd_GcdUnitDpath dpath
41   (
42     .*
43   );
44
45 endmodule
```

Figure 38: 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.

Figure 39: Verifying GCD Using Test Sources and Sinks – Parameterized test sources send a stream of messages over a val/rdy interface, and parameterized test sinks receive a stream of messages over a val/rdy interface and compare each message to a previously specified reference message. Clock and reset signals are not shown.

```
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 40: 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.

We can use the exact same source/sink testing that we used in the PyMTL3 tutorial to test our Verilog implementation of the GCD unit. As a reminder, Figure 39 illustrates the overall connectivity in the test harness. The test source includes the ability to delay messages going into the DUT and the test sink includes the ability to apply back-pressure to the DUT. By using various combinations of these delays we can more robustly ensure that our flow-control logic is working correctly. We can reuse the same test harnesses from the PyMTL3 tutorial as follows:

```
% cd ${TUTROOT}/build
% pytest ../tut4_verilog/gcd/test/GcdUnitRTL_test.py -v
% pytest ../tut4_verilog/gcd/test/GcdUnitRTL_test.py -sv -k basic_0x0
```

Figure 40 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.

★ *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.

6.2. Evaluating 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 simulators and look at the average number of cycles to compute a GCD for each input dataset like this:

```
% cd ${TUTROOT}/build
% ../tut4_verilog/gcd/gcd-sim --stats --impl cl  --input random
% ../tut4_verilog/gcd/gcd-sim --stats --impl rtl --input random
```

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.