In the lab assignments for this course, we will be using the Verilog hardware description language to model processors, memories, and networks at the register-transfer level (RTL). This tutorial briefly reviews the basics of the Verilog hardware description language, but primarily focuses on the specific Verilog development, testing, and evaluation framework as well as the coding conventions we will be using in the course. We will be using two primary open source tools for the lab assignments: Icarus Verilog (iverilog) for compiling Verilog models into simulators; and GTKWave (gtkwave) for viewing waveforms. Both tools are installed and available on amdpool. This tutorial assumes that students have completed the Linux and Git tutorials.

To follow along with the tutorial, access the ECE computing resources, and type the commands without the % character. 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 ece4750-setup script or that you have added it to your .bashrc script, which will then source the script every time you login. Sourcing the setup script sets up the environment required for this class.

**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 ECE computing resources. All of the code for the tutorial is located here: https://github.com/cbatten/ece4750-tut3-verilog. You will not use the ece4750-lab-admin script, and your specific environment may be different from what is assumed in this tutorial.

### 1. Verilog Modeling: Synthesizable vs. Non-Synthesizable

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 test harnesses, assertions, and line tracing. **So students must be very diligent in actively deciding whether or not they are writing synthesizable register-transfer-level models or non-synthesizable code.**

Students' design work will almost exclusively use synthesizable register-transfer-level (RTL) models, while students' verification work will likely use both synthesizable and non-synthesizable constructs. There is nothing wrong in using non-synthesizable constructs in a test harness since the test harness is not meant to model hardware just test hardware. Similarly, it is acceptable to include a limited amount of non-synthesizable code in students' designs for the sole purpose of debugging, assertions, and 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.
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. Note that pretty much any construct is allowed when implementing test harnesses for verification. There are also no limits on using the Verilog preprocessor, since the preprocessing step happens at compile time.

2. 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 directory to work in.

```
% source setup-ece4750.sh
% cd ${HOME}
% mkdir ece4750-tut3-verilog
% cd ece4750-tut3-verilog
```

2.1. 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 that we can then use to test the model’s functionality and evaluate the model’s performance. You can run iverilog as follows to compile `hello-world.v`.

```
% cd ${HOME}/ece4750-tut3-verilog
% 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 ${HOME}/ece4750-tut3-verilog
% ./hello-world
```

```
module top;
initial begin
    $display( "Hello World!" );
end
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

<table>
<thead>
<tr>
<th>&amp;</th>
<th>bitwise AND</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bitwise OR</td>
</tr>
<tr>
<td>~</td>
<td>bitwise XOR</td>
</tr>
<tr>
<td>~~</td>
<td>bitwise XNOR</td>
</tr>
<tr>
<td>~</td>
<td>bitwise NOT</td>
</tr>
</tbody>
</table>

| &&  | boolean AND  |
| ||  | boolean OR   |
| !   | boolean NOT  |

Arithmetic Operators

| +   | addition     |
| -   | subtraction  |

Reduction Operators

| &   | reduce via AND  |
|    | reduce via NAND |
| l   | reduce via OR   |
| ~   | reduce via NOR  |
| ~ ~ | reduce via XNOR |

Shift Operators

| >>  | shift right     |
| <<  | shift left      |
| >>> | arithmetic shift right |

Relational Operators

| ==  | equal          |
| !=  | not equal      |
| >   | greater than   |
| >=  | greater than or equals |
| <   | less than      |
| <=  | less than or equals |

Other Operators

| {}  | concatenate    |
| {N{}} | replicate N times |

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

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

```bash
% cd ${HOME}/ece4750-tut3-verilog
% 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.

2.2. Logic Data Types

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

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

// Declare single-bit logic variables.
logic a;
logic b;
logic c;

initial begin

// Single-bit literals
a = 1'bo; $display( "1'bo = %x ", a );
a = 1'b1; $display( "1'b1 = %x ", a );
a = 1'bX; $display( "1'bX = %x ", a );
a = 1'bZ; $display( "1'bZ = %x ", a );

// Bitwise logical operators for doing AND, OR, XOR, and NOT
a = 1'bo; b = 1'b1;
c = a & b; $display( "0 & 1 = %x ", c );
c = a | b; $display( "0 | 1 = %x ", c );
c = a ^ b; $display( "0 ^ 1 = %x ", c );
c = ~b; $display( "~1 = %x ", c );

// Bitwise logical operators for doing AND, OR, XOR, and NOT with X
a = 1'bo; b = 1'bX;
c = a & b; $display( "0 & x = %x ", c );
c = a | b; $display( "0 | x = %x ", c );
c = a ^ b; $display( "0 ^ x = %x ", c );
c = ~b; $display( "~x = %x ", c );

// Boolean logical operators
a = 1'bo; b = 1'b1;
c = a && b; $display( "0 && 1 = %x ", c );
c = a || b; $display( "0 || 1 = %x ", c );
c = !b; $display( "!1 = %x ", c );
end
endmodule

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

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

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

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

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

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

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

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

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

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

Lines 45–50 illustrate the reduction operators. These operators take a multi-bit logic value and com-
bine 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
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);

end
endmodule


Figure 3: Verilog Basics: Multi-Bit Logic and Logical Operators – Experimenting with multi-bit logic variables and literals, bitwise logical operators, and reduction operators.
2.3. 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.

```verilog
module top;

logic [7:0] A;
logic [7:0] B;
logic [7:0] C;

initial begin

// Fixed shift amount for logical shifts
A = 8'b1110_0101;
C = A << 1; $display( "8'b1110_0101 << 1 = %b", C );
C = A << 2; $display( "8'b1110_0101 << 2 = %b", C );
C = A << 3; $display( "8'b1110_0101 << 3 = %b", C );

// Fixed shift amount for arithmetic shifts
A = 8'b0110_0100;
C = $signed(A) >>> 1; $display( "8'b0110_0100 >>> 1 = %b", C );
C = $signed(A) >>> 2; $display( "8'b0110_0100 >>> 2 = %b", C );
C = $signed(A) >>> 3; $display( "8'b0110_0100 >>> 3 = %b", C );

// Variable shift amount for logical shifts
A = 8'b1110_0101;
B = 8'd2;
C = A << B; $display( "8'b1110_0101 << 2 = %b", C );
C = A >> B; $display( "8'b1110_0101 >> 2 = %b", C );

end
endmodule
```


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

7
2.4. 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?

```verilog
module top;

logic [7:0] A;
logic [7:0] B;
logic [7:0] C;

initial begin

// Basic arithmetic with no overflow or underflow
A = 8’d28;
B = 8’d15;
C = A + B; $display( "8’d28 + 8’d15 = %d", C );
C = A - B; $display( "8’d28 + 8’d15 = %d", C );

// Basic arithmetic with overflow and underflow
A = 8’d250;
B = 8’d15;
C = A + B; $display( "8’d250 + 8’d15 = %d", C );
C = B - A; $display( "8’d15 - 8’d250 = %d", C );

end
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.
2.5. 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; // 15 = 10011 in binary
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; // 4'b10xx
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 two's complement
B = 4’d0001; // 1 in two's 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


Figure 6: Verilog Basics: Relational Operators – Experimenting with relational operators.
2.6. 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.

```
module top;

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 [7:0] D; // 18-bit logic variable
logic [11:0] E; // 12-bit logic variable

initial begin

    // Basic concatenation
    A = 4’hA; B = 4’hB; C = 4’hC;
    D = 8’hde;

    E = { A, B, C }; $display("{ 4’hA, 4’hB, 4’hC } = %x", E);
    E = { C, A, B }; $display("{ 4’hC, 4’hA, 4’hB } = %x", E);
    E = { B, C, A }; $display("{ 4’hB, 4’hC, 4’hA } = %x", E);

    E = { A, D }; $display("{ 4’hA, 8’hde } = %x", E);
    E = { D, A }; $display("{ 8’hde, 4’hA } = %x", E);

    E = { A, 8’hf0 }; $display("{ 4’hA, 8’hf0 } = %x", E);
    E = { 8’hf0, A }; $display("{ 8’hf0, 4’hA } = %x", E);

    // Repeat operator
    A = 4’hA;
    B = 4’hB;

    E = { 3{A} }; $display("{ 4’hA, 4’hA, 4’hA } = %x", E);
    E = { A, {2{B}} }; $display("{ 4’hA, 4’hB, 4’hB } = %x", E);

end
endmodule
```


Figure 7: Verilog Basics: Concatenation Operators – Experimenting with the basic concatenation operator and the repeat operator.
2.7. 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 label 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.
2.8. Struct Data Types

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 your own new struct type for holding the request messages we will use in the GCD example later in this tutorial. The new struct type should be called gcd_req_msg_t and it should include two fields: a 32-bit field named a and a 32-bit field named b. Write some code to write and read these 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.
2.9. 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. 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:

```verilog
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:

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

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

---

```
module top;
logic [7:0] a;
logic [7:0] b;
logic [7:0] c;
logic [1:0] sel;
initial begin
// ternary statement
a = 8'd30;
b = 8'd16;
c = ( a < b ) ? 15 : 14;
$display( "c = %d", c );
c = ( b < a ) ? 15 : 14;
$display( "c = %d", c );
// nested ternary statement
sel = 2'b01;
c = ( sel == 2'b00 ) ? 8'h0a :
( sel == 2'b01 ) ? 8'h0b :
( sel == 2'b11 ) ? 8'h0d :
8'h0e;
$display( " sel = 1, c = %b", c );
// nested ternary statement w/ X
sel = 2'bxx;
c = ( sel == 2'b00 ) ? 8'h0a :
( sel == 2'b01 ) ? 8'h0b :
( sel == 2'b11 ) ? 8'h0d :
8'h0e;
$display( " sel = x, c = %b", c );
sel = 2'bx0;
c = ( sel == 2'b00 ) ? 8'h0a :
( sel == 2'b01 ) ? 8'h0b :
( sel == 2'b11 ) ? 8'h0d :
8'h0e;
$display( " sel = x, c = %b", c );
end
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.
2.10. 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.

```verilog
module top;
logic [7:0] a;
logic [7:0] b;
logic sel;
initial begin
  // if statement
  a = 8'd30;
b = 8'd16;
  if ( a == b ) begin
    $display( "30 == 16" );
  end
  if ( a != b ) begin
    $display( "30 != 16" );
  end
  // if else statement
  sel = 1'b1;
  if ( sel == 1'b0 ) begin
    a = 8'h0a;
  end
  else begin
    a = 8'h0b;
  end
  $display( " sel = 1, a = %x ", a );
  // if else statement w/ X
  sel = 1'bx;
  if ( sel == 1'b0 ) begin
    a = 8'h0a;
  end
  else begin
    a = 8'h0b;
  end
  $display( " sel = x, a = %x ", a );
endmodule
```

Figure 11: Verilog Basics: If Statements — Experimenting with if statements including what happens if the conditional includes an unknown.
2.11. 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.

```
module top;

// Declaring Variables
logic [1:0] sel;
logic [7:0] a;

initial begin
  // case statement
  sel = 2'b01;
  case ( sel )
    2'b00 : a = 8'h0a;
    2'b01 : a = 8'h0b;
    2'b10 : a = 8'h0c;
    2'b11 : a = 8'h0d;
    default : a = 8'h0e;
  endcase
  $display( "sel = 01, a = %x", a );
  // case statement w/ X
  sel = 2'bxx;
  case ( sel )
    2'b00 : a = 8'h0a;
    2'b01 : a = 8'h0b;
    2'b10 : a = 8'h0c;
    2'b11 : a = 8'h0d;
    default : a = 8'h0e;
  endcase
  $display( "sel = xx, a = %x", a );
  // Do not use x's in the case
  // selection items
  sel = 2'bx0;
  case ( sel )
    2'b00 : a = 8'h0a;
    2'b01 : a = 8'h0b;
    2'b10 : a = 8'h0c;
    2'b11 : a = 8'h0d;
    default : a = 8'h0e;
  endcase
  $display( "sel = x0, a = %x", a );
end
endmodule
```

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.
2.12. 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?

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

Code at https://github.com/cbatten/x/blob/master/ex-basics-casez.v
3. Introduction to the Modular Verilog Build System

In the previous section, all Verilog source files were in a single directory, we manually ran `iverilog` to build each small test, and the directory contained a mix of source files and compiled simulators. While this is fine for very small projects, by the final lab we will have hundreds of Verilog source files. Larger, more complicated projects require some kind of more structured “build system” to organize Verilog source files, build simulators, and run these simulators for testing or evaluation.

In this course, we will be using a custom modular Verilog build system developed by the course instructors. Use the following commands to checkout the example project we will be working on in this tutorial:

```
% source setup-ece4750.sh
% ece4750-lab-admin start ece4750-tut3-verilog
% cd ${HOME}/ece4750/ece4750-tut3-verilog
% TUTROOT=${PWD}
```

The modular Verilog build system is based on the concept of a project containing a collection of subprojects. The subprojects are meant to be modular in the sense that they should be relatively self-contained and should explicitly indicate any dependencies on other subprojects. Take a few minutes to look at the files that are included in this project:

- `COPYING` – Open-source license
- `Makefile.in` – Input used to create makefile for project
- `aclocal.m4` – Autoconf fragment used by `configure.ac`
- `configure.ac` – Input used to create configure script for project
- `configure` – Configure script used to configure the project
- `subprojs.mk` – List of subprojects that make up this project
- `scripts` – Scripts used by the build system
- `vc` – Subproject containing common Verilog components
- `ex-basics` – Subproject containing basic code from previous section
- `ex-sorter` – Subproject containing example sorting unit
- `ex-gcd` – Subproject containing example greatest-common-divisor unit

For the most part, you will not need to modify nor really understand any of the files at the top-level of the project; the only exception is the `subprojs.mk` makefile fragment that you will use to explicitly specify which subprojects make up the project.

The basic process for building and testing your project should be familiar to anyone that has built open-source software from source. You will need to create a build directory, configure the project, build the project, and then run the tests. We do this with the following set of commands:

```
% cd ${TUTROOT}
% mkdir build
% cd build
% ../configure
% make
% make check
```

The configure script will check various aspects of the system you are working on to make sure it is setup correctly. For example, the configure script will make sure that your system has an acceptable Verilog compiler installed. When the configure script is finished it will create a new makefile in the build directory (based on the `Makefile.in` file located at the top of the project) which you can use...
to build, test, and evaluate your project. The make command uses the makefile to figure out how to "make" various aspects of the project. By default, the make command will build all of the Verilog simulators we use for evaluation. We can pass various "targets" to more specifically tell Make what to do; for example, the check target tells Make to run all of the unit tests. You should see no failures for the tests in the vc subproject, and you will see failures for the ex-sorter subproject that we will fix later. We will be using make to drive our hardware development toolflow.

3.1. Creating a New Subproject

As mentioned above, each subproject should be relatively self-contained and encapsulate one relatively large part of the design. For example, each of the lab assignments will be a separate subproject, and in the final lab assignment we will leverage all of these subprojects to create a new subproject that contains the full multicore system. Normally, the lab harness you receive will have a subproject already created for you to use when working on the baseline and alternative design. Any extensions you develop should be in a separate subproject to make it easier for the course staff to evaluate your baseline and alternative designs.

We will now create a new subproject for use in the next section of the tutorial. A subproject is really just a set of policies, naming conventions, and a few special files. Each subproject needs a name which will be used as a prefix in many different ways. Subproject names should be short, lower-case, and only contain letters, numbers, and dashes (-). Our new subproject will be named ex-regincr (ex stands for PARC Example). We will use the subproject name (or a variation of the subproject name) as a prefix for subproject filenames (e.g., ex-regincr-), Verilog macro names (e.g., EX_REGINCR_), Verilog module names (e.g., ex_regincr_), and Make variable names (e.g., ex_regincr_). To reiterate, every subproject filename, Verilog macro name, Verilog module name, and Make variable name must use the subproject name as a prefix. This is essential to avoid namespace collisions with other subprojects.

Each subproject has its own dedicated subdirectory named after the subproject. Create a subdirectory for our new subproject as follows:

```
% cd ${TUTROOT}
% mkdir ex-regincr
```

The key way that the build system knows about the dependencies and files of the subproject is through a special makefile fragment which goes in the subproject’s subdirectory. The special makefile fragment should always have the same name as the subproject with a .mk filename extension. This makefile fragment should have at least the following four Make variables (where subproj is the name of the subproject with dashes replaced by underscores).

- subproj_deps – List of other subprojects this subproject depends on
- subproj_srcs – List of Verilog source files with .v filename extension
- subproj_test_srcs – List of Verilog unit test source files with .t.v filename extension
- subproj_sim_srcs – List of Verilog simulator source files with .v filename extension

Let’s go ahead and create a makefile fragment for our new subproject, but we will leave all of the variables empty for now.

```
% cd ${TUTROOT}/ex-regincr
% echo "
ex_regincr_deps =
ex_regincr_srcs =
ex_regincr_test_srcs =
ex_regincr_sim_srcs =
```

21
The final step in creating a new subproject to update the build system’s master list of subprojects. This master list is contained in the `subprojs.mk` file at the top of the project. If you look in `subprojs.mk` you will see that it currently lists the `vc`, `ex-sorter`, and `ex-gcd` subprojects. Let’s add our new subproject to this list:

```bash
% cd ${TUTROOT}
% echo "subprojs = vc ex-basics ex-regincr ex-sorter" > subprojs.mk
```

We can now go ahead and try rebuilding the project:

```bash
% cd ${TUTROOT}/build
% make
% make echo-subprojs
```

If you added your project correctly, then this should list all of the subprojects including the new `ex-regincr` subproject. Since we haven’t actually added any new Verilog source files the build system won’t really do anything different; but we are now ready to add our first Verilog hardware module in the next section of the tutorial.

### 3.2. Primary Makefile Targets

A few other makefile targets included as part of the modular Verilog build system are listed below.

- `all` – build all evaluation simulators in the project
- `check` – build and run all unit tests in the project
- `clean` – clean all generated content in the project
- `sproj-all` – build all evaluation simulators for the subproject named `sproj`
- `sproj-check` – run all unit tests for the subproject named `sproj`
- `sproj-eval` – evaluate all simulators for the subproject named `sproj`
- `sproj-clean` – clean all generated content for the subproject named `sproj`

For example, the following commands will run all the unit tests for just the `vc` subproject and then clean all generated content for the `vc` subproject.

```bash
% cd ${TUTROOT}/build
% make vc-check
% make vc-clean
```

### 4. Registered Incrementer: Behavioral Verilog Development and Verification

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. 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.
Figure 14: Block Diagram for Registered Incrementer – An eight-bit registered +2 incrementer with an eight-bit input port, an eight-bit output port, and an (implicit) clock input.

```verilog
// ===========================================================================
// Registered Incrementer
// ===========================================================================
// This is a simple example of a module for a registered incrementer
// which combines a positive edge triggered register with a combinational
// +2 incrementer. We use flat register-transfer-level modeling.

ifndef EX_REGINCR_REG_INCR_V

define EX_REGINCR_REG_INCR_V

module ex_regincr_RegIncr
(
  input logic clk,
  input logic [7:0] in,
  output logic [7:0] out
);

// Sequential logic
logic [7:0] temp_reg;
always @(posedge clk )
  temp_reg <= in;

// Combinational logic
logic [7:0] temp_wire;
always @(*)
  temp_wire = temp_reg + 1;

// Combinational logic
assign out = temp_wire + 1;
endmodule

endif /* EX_REGINCR_REG_INCR_V */
```

Code at https://github.com/cbatten/x/blob/master/ex-regincr-RegIncr.v

Figure 15: Registered Incrementer – An eight-bit registered +2 incrementer corresponding to the diagram in Figure 14.

4.1. RTL Behavioral Model of Registered Incrementer

Figure 15 shows the Verilog code which corresponds to the diagram in Figure 14. Every Verilog file should begin with a header comment as shown on Lines 1–6 in Figure 15. The header comment should identify the primary module in the file, and include a brief description of what the module does. Reserve discussion of the actual implementation for later in the file. In general, you should attempt to keep lines in your Verilog source code to less than 74 characters. This will make your code easier to read, enable printing on standard sized paper, and facilitate viewing two source files side-by-side on a single monitor. Lines 8–9 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 36.
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 11–16 in Figure 15 illustrate how we represent this interface using Verilog. A common mistake is to forget the semicolon (; ) on Line 16. 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 13–15, 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 12 and 16, 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 ).

Lines 17–33 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 @ ( posedge clk ) concurrent blocks to model sequential logic and always @ (*) concurrent blocks to model combinational logic. We should always use the newer @(*) syntax and never use explicit sensitivity lists. 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 22) and we should only use blocking assignments in combinational logic (e.g., the = operator on Line 28). We use the variable temp_reg to hold the intermediate value between the register and the first +1 incrementer, and we use the variable temp_wire to hold the intermediate value between the two +1 incrementers. temp_reg 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 temp_reg 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 @(*) concurrent block to model the first +1 incrementer and a continuous assignment statement (i.e., assign ) to model the second +1 incrementer. 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 @(*) 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 @(*) 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.
Create a new Verilog source file named `ex-regincr-RegIncr.v` that contains the code in Figure 15 using your favorite text editor. You should place this file in the `ex-regincr` subproject and add it to the list of files in the `ex-regincr.mk` makefile fragment like this:

```bash
% cd ${TUTROOT}/ex-regincr
% echo "
ex_regincr_deps =
ex_regincr_srcs = ex-regincr-RegIncr.v
ex_regincr_test_srcs =
ex_regincr_sim_srcs =
" > ex-regincr.mk
```

### 4.2. Verifying the Registered Incrementer Using Ad-Hoc Unit Testing

Now that we have developed a new hardware module, your first thought should always turn to testing that module. Figure 16 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 7 includes the Verilog source file that contains the registered incremeneter. Lines 13–14 setup a clock with a period of 10 time steps. Notice that we are assigning an initial value to the `clk` net on Line 13 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 18–26 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 23–25 illustrate the correct coding convention with one port binding per line and the ports/nets vertically aligned. As shown on Lines 22 and 26 the opening and closing parenthesis should be on their own separate lines. Although this may seem verbose, this coding style can significantly reduce errors by making it much easier to quickly visualize how ports are connected.

Lines 30–64 illustrate an `initial` block which executes at the very beginning of the simulation. `initial` blocks are not synthesizable and should only be used in test harnesses. Lines 34–35 instruct the simulator to dump waveforms for all nets. Line 37 is a delay statement that essentially waits for one timestep. Delay statements are not synthesizable and should only be used in test harnesses. Lines 41–46 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.

Create a new Verilog source file named `ex-regincr-RegIncr-adhoc-test.v` that contains the code in Figure 16 using your favorite text editor. You should place this file in the `ex-regincr` subproject and add it to the list of files in the `ex-regincr.mk` makefile fragment like this:

```bash
% cd ${TUTROOT}/ex-regincr
% echo "
ex_regincr_deps =
ex_regincr_srcs = ex-regincr-RegIncr.v
ex_regincr_test_srcs =
ex_regincr_sim_srcs = ex-regincr-RegIncr-adhoc-test.v
" > ex-regincr.mk
```
// RegIncr Ad-Hoc Testing
// This is an example of ad-hoc testing which does not use any specific
// unit testing framework.

module top;

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

// Instantiate the device-under-test
logic [7:0] in;
logic [7:0] out;

ex_regincr_RegIncr reg_incr
(
.clk (clk),
in (in),
.out (out)
);

// Verify functionality
initial begin

// Dump waveforms
$dumpfile("ex-regincr-RegIncr-test.vcd");
$dumpvars;
#1;

// Test cases
in = 8'h00;
#10;
if ( out != 8'h02 ) begin
    $display("ERROR: out, expected = %x, actual = %x", 8'h02, out);
    $finish;
end
in = 8'h13;
#10;
if ( out != 8'h15 ) begin
    $display("ERROR: out, expected = %x, actual = %x", 8'h15, out);
    $finish;
end
in = 8'h27;
#10;
if ( out != 8'h29 ) begin
    $display("ERROR: out, expected = %x, actual = %x", 8'h29, out);
    $finish;
end
$display("*** PASSED ***");
$finish;
endmodule


Figure 16: Ad-Hoc Test Harness for Registered Incrementer – An ad-hoc test harness for the eight-bit registered incremeneter in Figure 15.
We are now finally ready to test our registered incrementer module. You can build and run the ad-hoc test as follows:

```bash
% cd ${TUTROOT}/build
% make ex-regincr-RegIncr-adhoc-test
% ./ex-regincr-RegIncr-adhoc-test
```

If you take a close look at the output from running make, you should see make using iverilog to build the ad-hoc test. Essentially, make is automating the manual process we used in the previous section. You might also note that make uses a helper script called warnings2errors. Because Verilog is a very permissive language, iverilog will compile code which while technically allowed is almost certainly wrong with just a warning. The warnings2errors helper script will convert all warnings into errors and halt the compilation process. This forces the designer to eliminate all warnings and should help avoid a large class of subtle bugs. 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 +4 instead of +2. 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 +4 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 +2.

4.3. Verifying the Registered Incrementer Using Unit Testing Framework

There are many issues with using ad-hoc test harnesses similar to what is shown in Figure 17. Ad-hoc testing is verbose, which makes it error prone and more cumbersome to write tests. Ad-hoc testing is difficult for others to read and understand since by definition it is ad-hoc. Ad-hoc testing does not use any kind of standard test output, and does not provide support for controlling the amount of test output. In this course, we will be using a simple unit testing framework to make it easier to write, maintain, and run unit tests. This framework is in `vc-test.v` as part of the vc subproject.

Figure 17 shows how we can use this simple framework to unit test our registered incrementer. Note that we need to explicitly include `vc-test.v` at the top of our unit test. All of our unit tests should be in a single module named `top`. The `VC_TEST_SUITE_BEGIN` macro on Line 9 is used to set the name of this test suite and initialize the framework; note that we must include a corresponding `VC_TEST_SUITE_END` right before the end of the module.

Lines 11–54 illustrate a single test case; notice that all net names, module instance names, and tasks are prefixed with `t1` corresponding to test case number one. Our unit tests will usually include several test cases, and each test case should use a unique prefix (e.g., `t2`, `t3`, etc.). The test case has three parts: Lines 15–23 instantiate the device-under-test (DUT), Lines 27–39 declare a helper task, and Lines 43–54 use the helper task to test the DUT. We should use a similar coding convention as described above for instantiating the DUT. All input ports should be connected to `reg` nets and all output ports should be connected to `wire` nets. The helper task should include one input for each port of the DUT, and it should first set all of the inputs for the DUT and then use macros from `vc-test.v` to verify that the outputs from the DUT are as expected. The delay statements on Lines 34, 37, and 46 are used to ensure that we set inputs and verify outputs right after the rising clock edge. The `VC_TEST_NOTE_INPUTS_1` macro is used to display the values of all inputs to the DUT and can be useful when trying to debug failing test cases. There are corresponding macros which take two, three, and four arguments. The `VC_TEST_NET` macro is used to verify that a net has an expected value. The actual test case is declared using `VC_TEST_CASE_BEGIN` and `VC_TEST_CASE_END` macros.
/* ex - regincr - RegIncr Unit Tests */

module top;  
  'VC_TEST_SUITE_BEGIN( "ex - regincr - RegIncr" )
  // Test ex - regincr - RegIncr
  logic [7:0] t1_in;
  logic [7:0] t1_out;
  ex_regincr_RegIncr t1_reg_incr
    (. clk ( clk ),
     . in ( t1_in ),
     . out ( t1_out )
    );
  // Helper task
  task t1
    ( input logic [7:0] in,
      input logic [7:0] out
    );
    begin
      t1_in = in;
      #1;
      'VC_TEST_NOTE_INPUTS_1( in );
      'VC_TEST_NET( t1_out, out );
      #9;
    end
  endtask
  // Test case
  'VC_TEST_CASE_BEGIN( 1, "simple" )
  begin
    #1;
    t1( 8'h00, 8'h?? );
    t1( 8'h13, 8'h02 );
    t1( 8'h27, 8'h15 );
    t1( 8'hxx, 8'h29 );
  end
  'VC_TEST_CASE_END
  'VC_TEST_SUITE_END
endmodule

Code at https://github.com/cbatten/x/blob/master/ex-regincr-RegIncr.t.v

Figure 17: Unit Tests for Registered Incrementer – A set of unit tests for the eight-bit registered incrementer in Figure 15, which uses our vc-Test unit testing framework.
You must explicitly pass the test case number to `VC_TEST_CASE_BEGIN`, and a common mistake is to use the same test case number more than once. We call the helper task once per line, and this basically means each line of our test corresponds to setting the DUT inputs and verifying the DUT outputs for one cycle. Note that you can use an `x` to indicate inputs which you do not care about (e.g., Line 51) and you can use a question mark (`?`) to indicate outputs for which any value is acceptable (e.g., Line 48).

The unit testing in Figure 17 is an example of directed cycle-by-cycle gray-box testing. It is directed since we are explicitly creating directed tests as opposed to using some kind of random testing. It is cycle-by-cycle since we are explicitly setting the inputs and verifying the outputs every cycle. Black-box testing describes a testing strategy where the test cases depend only on the interface and not the specific implementation of the DUT (i.e., they should be valid for any correct implementation). White-box testing describes a testing strategy where the test cases depend on the specific implementation of the DUT (i.e., they may not be valid for every correct implementation). The test cases in Figure 17 are black-box with respect to the functional behavior of the DUT but they are white-box with respect to the timing behavior of the device. The test cases rely on the fact that the registered incrementer includes exactly one edge and they would fail if we pipelined the incrementer such that each transaction took two edges. In Section 6, we will see how we can use latency-insensitive interfaces to create true black-box unit tests.

Create a new Verilog source file named `ex-regincr-RegIncr.t.v` that contains the code in Figure 17 using your favorite text editor. Note that all unit tests must use the `.t.v` filename extension. You should place this file in the `ex-regincr` subproject and add it to the list of files in the `ex-regincr.mk` makefile fragment like this:

```bash
% cd ${TUTROOT}/ex-regincr
ex_regincr_deps =
ex_regincr_srcs = ex-regincr-RegIncr.v
ex_regincr_test_srcs = ex-regincr-RegIncr.t.v
ex_regincr_sim_srcs =
" > ex-regincr.mk
```

We can build and run our unit tests as follows:

```bash
% cd ${TUTROOT}/build
% make ex-regincr-RegIncr-test
% ./ex-regincr-RegIncr-test
```

Our simple unit testing framework provides support for verbose output which displays all tests even if they pass with the `+verbose` command-line option. You can specify a specific test case to run with the `+test-case` command-line option.

```bash
% cd ${TUTROOT}/build
% ./ex-regincr-RegIncr-test +verbose=1 +test-case=1
```

Developers will spend a great deal of time editing Verilog source code, rebuilding a unit test, and then rerunning this unit test. We can simplify this `edit-compile-test` development cycle by building a unit test and running that test on a single command line as follows:

```bash
% cd ${TUTROOT}/build
% make ex-regincr-RegIncr-test && ./ex-regincr-RegIncr-test
```
This way we can make an edit in our editor, change to our terminal, press the up-arrow key, and press return. Remember that you can run all unit tests in a given subproject or all unit tests in the entire project using the following make targets:

```bash
% cd ${TUTROOT}/build
% make ex-regincr-check
% make check
```

We recommend you spend most of your time building and running a single unit test when debugging your hardware models, since this is faster and enables you to easily change the verbosity or run a specific test case. Reserve the makefile check targets for when you want to quickly verify that everything is working.

**To-Do On Your Own:** Edit the register incrementer so that it now increments by +4 instead of +2. 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 +4 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 +2.

### 4.4. Debugging the Registered Incremenetr Using Waveforms

It is sometimes possible to debug your design purely using the output from your unit tests, but often we need to inspect more signals than are displayed through the unit testing framework. While we could add calls to the `$display()` system task to output more and more signals in our design, it is much more productive to use the built-in capabilities of the Verilog tools to generate waveforms for each net in your design. We will be using the Verilog Change Dump (VCD) format for our waveforms.

To have a unit test generate VCD, you need to pass it the `+dump-vcd` command line option. This will cause the unit test framework to generate a VCD file with the same name as the test and a `.vcd` file name extension.

```bash
% cd ${TUTROOT}/build
% ./ex-regincr-RegIncr-test +dump-vcd
```

You can use the open-source GTKWave program to browse the generated waveforms as follows:

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

You can browse the module hierarchy of your design in the upper-left panel, with the signals in any given module being displayed in the lower-left panel. Select signals and use the `Append` or `Insert` button to add them to the waveform panel on the right. You can drag-and-drop signals to arrange them as desired. To see the full hierarchical names of each signal choose `Edit > Toggle Trace Hierarchy` or simply press the H key. Choose `File > Reload Waveform` (or press the blue circular arrows in the toolbar) to update GTKWave after you have rerun a simulation. Organizing signals can sometimes be quite consuming, so you can save and load the current configuration using `File > Write Save File` and `File > Read Save File`. Figure 18 illustrates using GTKWave to view the waveforms from our ad-hoc test. GTKWave has many useful options which can make debugging your design more productive, so feel free to explore the associated documentation.
Figure 18: GTKWave Waveform Viewer – GTKWave is being used to browse the signals associated with the registered incrementer shown in Figure 15 and the ad-hoc test shown in Figure 17.

Note that viewing waveforms should never take the place of automated verification through the unit testing framework. Students might be tempted to simply look at the waveforms to determine if their design is working, but this kind of “verification by inspection” is error prone and not reproducible. If you later make a change to your design, you would have to take another look at the waveforms to ensure that your design still works. If another member of your group wants to understand your design and verify that it is working, he or she would also need to take a look at the waveforms. While this might be feasible for very simple designs, it is obviously not a scalable approach when building the more complicated designs we will tackle in this course. Automated testing using the unit testing framework is the best way to permanently verify your design.

To-Do On Your Own: Edit the register incrementer so that it now increments by +4 instead of +2. Recompile, rerun the unit test (with the +dump-vcd option), and take another look the waveforms to see how they have changed.

5. Sorter: Structural Verilog Development and Verification

In this section, we will explore slightly more advanced techniques including parameterized design, line tracing, structural composition, randomized testing, and evaluating the performance of a design with a simulator. We will be using the simple pipelined four-element sorter shown in Figure 19 as an example. The code related to this part of the tutorial is located in the ex-sorter subproject. 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.

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

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 ex-sorter-SorterFlat.v. Figure 20 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 21 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 unit test is in ex-sorter-SorterFlat.t.v, although if you look in this file you will see that it only contains a few lines. We are often evaluating several different implementations of a module, and we would like to be able to easily run all of the same tests on all implementations. To reduce duplicate code, we have refactored the bulk of the test code into a single file called ex-sorter-test-harness.v. We use a special pre-processor macro when instantiating the DUT named EX_SORTER_IMPL in ex-sorter-test-harness.v. Then to test an implementation we simply define this special macro, include the Verilog file that contains the implementation, and include the Verilog file that contains the test harness. You build and run the unit tests as follows:

```
% cd ${TUTROOT}/build
% make ex-sorter-SorterFlat-test && ./ex-sorter-SorterFlat-test
```

The design should pass all of the tests. We are using a very similar strategy as in the previous section based on directed cycle-by-cycle gray-box testing. Since the sorter includes an output valid bit, it
module ex_sorter_SorterFlat
#(
    parameter p_nbits = 1
)
(
    input logic clk ,
    input logic reset ,
    input logic in_val ,
    input logic [p_nbits-1:0] in0 ,
    input logic [p_nbits-1:0] in1 ,
    input logic [p_nbits-1:0] in2 ,
    input logic [p_nbits-1:0] in3 ,
    output logic out_val ,
    output logic [p_nbits-1:0] out0 ,
    output logic [p_nbits-1:0] out1 ,
    output logic [p_nbits-1:0] out2 ,
    output logic [p_nbits-1:0] out3
);

Figure 20: Interface for the Four-Element Sorter – The interface corresponds to the diagram in Figure 19 and is parameterized by the bitwidth of each element.

should be possible to use a black-box testing strategy by waiting for the valid bit to be true before checking the outputs. This would allow the same test harness to test various implementations that each take a different number of cycles to sort the elements (e.g., pipelined, iterative, single-cycle).

★ 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. Visualizing Execution with Line Traces

We have already learned about two methods to observe the operation of our device-under-test: the test output from our unit testing framework and the waveforms for every net in the design. The former is rather limited and the latter can sometimes be almost too detailed with bit accuracy and a multitude of nets to understand. As architects, we often want a way to visualize the execution of our device-under-test with an intermediate level of detail; for example, we want our models to generate high-level pipeline diagrams to illustrate how transactions are moving through a system.

In this course, we will use a technique called line tracing to achieve this goal. A line trace consists of plain-text trace output with each line corresponding to one (and only one!) cycle. Fixed-width columns will correspond to either state at the beginning of the corresponding cycle or the output of combinational logic during that cycle. Line traces will abstract the detailed bit representations of signals in our design into useful character representations. So for example, instead of visualizing instructions as raw bits, we will visualize them as text strings.

You can use the +trace command-line option to see a line trace of the four-element sorter.

% cd ${TUTROOT}/build
% ./ex-sorter-SorterFlat-test +test-case=1 +trace=1

Figure 22 shows a portion of the resulting 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 S0, S1, and S2 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
// Stage S0 -> S1 pipeline registers

logic val_S1;
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;

always @( posedge clk ) begin
val_S1 <= ( reset ) ? 0 : in_val;
elm0_S1 <= in0;
elm1_S1 <= in1;
elm2_S1 <= in2;
elm3_S1 <= in3;
end

// Stage S1 combinational logic

logic [p_nbits -1:0] elm0_next_S1;
logic [p_nbits -1:0] elm1_next_S1;
logic [p_nbits -1:0] elm2_next_S1;
logic [p_nbits -1:0] elm3_next_S1;

always @(*) begin

// Sort elms 0 and 1
if ( elm0_S1 <= elm1_S1 ) begin
elm0_next_S1 = elm0_S1;
elm1_next_S1 = elm1_S1;
end
else if ( elm0_S1 > elm1_S1 ) begin
elm0_next_S1 = elm1_S1;
elm1_next_S1 = elm0_S1;
end
else begin
elm0_next_S1 = 'hx;
elm1_next_S1 = 'hx;
end

// Sort elms 2 and 3
if ( elm2_S1 <= elm3_S1 ) begin
elm2_next_S1 = elm2_S1;
elm3_next_S1 = elm3_S1;
end
else if ( elm2_S1 > elm3_S1 ) begin
elm2_next_S1 = elm3_S1;
elm3_next_S1 = elm2_S1;
end
else begin
elm2_next_S1 = 'hx;
elm3_next_S1 = 'hx;
end

end

Figure 21: First Stage of the Flat Sorter Implementation – First pipeline stage of the sorter using a flat implementation corresponding to the diagram in Figure 19.
Figure 22: Example Line Trace for Sorter – 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.

```verilog
ifndef SYNTHESIS
reg [(\(VC_TRACE_NBITS_NCHARS(p_nbits)\)*4+5)*8] -1:0 str;

VC_TRACE_BEGIN
begin
// Inputs
$sformat(str, "\{0x,0x,0x,0x\}", in0, in1, in2, in3);
vc_trace.append_val_str(trace_str, in_val, str);
vc_trace.append_str(trace_str, "|");
...
end

VC_TRACE_END

'endif /* SYNTHESIS */
```

Figure 23: Example of Line Tracing Code – This code generates the first fixed-width column for the line trace shown in Figure 22

see when the hardware is actually doing useful work. The line trace shows the sorter working on a single sort and then trying multiple sorts at once in a pipelined fashion. You can see how each stage is swapping various elements, and you can see that if we push in elements fast enough we are able to keep the pipeline busy without any stalls. Line traces are a powerful way to visualize your design and debug both correctness and performance issues.

The VC library includes support in `vc-trace.v` to make it easier to create line traces in your own Verilog modules. Figure 23 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 our design. Line 3 declares a temporary string variable that we will use when converting nets into strings. Lines 5–17 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, S0, S1, S2, out) before each stage. So cycle 24 of the line trace shown in Figure 22 should now look like:

24: in:{05,07,08}|S0:{a5,a3,a2,a7}|S1:{03,01,02}|S2: |out:

5.3. Structural Sorter Implementation

The flat implementation in ex-sorter-SorterFlat.v is complex and monolithic and it fails to really exploit the structure inherent in the sorter. We can use modularity and hierarchy to divide complicated designs into smaller more manageable units; these smaller units are easier to design and can be tested independently before integrating them into larger, more complicated designs. We have started a structural implementation in ex-sorter-SorterStruct.v; the structural implementation will have an identical interface and behavior as the flat implementation in ex-sorter-SorterFlat.v.

Figure 24 shows the first pipeline stage of the structural implementation of the sorter. Our design instantiates three kinds of modules: vc_ResetReg, vc_Reg, and ex_sorter_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.

★ To-Do On Your Own: The structural implementation is incomplete because the actual implementation of the MinMax unit in vc-MinMaxUnit.v is not finished. You should go ahead and implement the MinMax unit, and then as always you should write a unit test to verify the functionality of your MinMax unit! You should have enough experience based on the previous sections to be able to create a unit test from scratch and add it to the build system. Once your MinMax unit is complete and tested, then test the structural sorter implementation like this:

% cd ${TUTROOT}/build
% make ./ex-sorter-SorterStruct-test && ./ex-sorter-SorterStruct-test
Figure 24: First Stage of the Structural Sorter Implementation – First pipeline stage of the sorter using a structural implementation corresponding to the diagram in Figure 19.
5.4. Verifying the Sorter Using Randomized Testing

So far we have mostly used a directed cycle-by-cycle gray-box testing strategy. Once we have finished writing hand-crafted directed tests, we almost always want to leverage randomized testing to further improve our confidence in the correct functionality of the design. Generating randomized test vectors in Verilog can be cumbersome, and indeed there are sophisticated programming languages whose sole purpose it is to facilitate more productive testing of hardware models. SystemVerilog adds comprehensive support for randomized testing, but these features are not yet supported in Verilog. In this course, we will take a simpler approach based on using Python to generate randomized test inputs and the corresponding correct outputs for verification. The build system simplifies creating Python scripts that output Verilog code to be included in our unit tests.

We have already included such a Python script in the ex-sorter subproject. Try running the script like this:

```
% cd ${TUTROOT}/build
% python ../ex-sorter/ex-sorter-gen-input.py random
```

Spend a few minutes looking at the source code for the Python script. The result will be a hundred test vectors written in a very similar style to what have been using in our directed tests. After writing a Python script like this, we need to tell the build system about the Verilog file we want to generate using this script. You can do this by adding a specially named Verilog file to a new make variable in the subprojects make fragment. Take a look at how the ex-sorter.mk file uses the ex_sorter_pyv_src make variable to specify three different Python-generated Verilog files. The build system uses the filename to figure out that to generate the Verilog file named ex-sorter-gen-input_random.py.v we need to run the ex-sorter-gen-input.py Python script with random as a command line argument. After updating the ex-sorter.mk makefile fragment you should be able to generate these Verilog files like this:

```
% cd ${TUTROOT}/build
% make ex-sorter-gen-input_random.py.v
```

We can include these generated Verilog files in dedicated test cases. Figure 25 illustrates a new test case we can add to ex-sorter-test-harness.v for a randomized testing strategy. Adding this new

```
// ----------------------------------------------------------------------
// Test Random
// ----------------------------------------------------------------------

```

Figure 25: Randomized Test Case for the Four-Element Sorter – The Verilog file ex-sorter-gen-input_random.py.v is generated by a Python script and contains 100 random test inputs and the corresponding correct outputs.
test case to the directed tests in `ex-sorter-test-harness.v` for both the flat and structural sorter implementations.

When you are done you can run all of the tests for our `ex-sorter` subproject like this:

```bash
% cd ${TUTROOT}/build
% make ex-sorter-check
```

**To-Do On Your Own:** Experiment with the two other random test vectors that the Python script can generate. Add two more random test cases to `ex-sorter-test-harness.v` which apply these two additional random test vectors to the sorter implementations. Recompile and rerun the unit tests.

### 5.5. Evaluating the Sorter Using a Simulator

So far we have focused on implementing and verifying our design, but our ultimate goal is to actually evaluate a design. We do not use unit tests for evaluation; instead we use a *evaluation simulator* which has been designed for quantitatively measuring the cycle-level performance of a specific implementation on a given input dataset. For this tutorial, we will create a simulator to compare the flat and structural implementations on the random datasets generated using the Python script described in the previous section.

Due to limitations of the Verilog language, we will need a separate evaluation simulator binary for every implementation we wish to evaluate. In order to reduce duplicate code, we refactor most of the code into a shared simulator harness which is analogous to the technique we used in creating a shared test harness. This is illustrated in the `ex-sorter-sim-harness.v`, `ex-sorter-sim-flat.v`, and `ex-sorter-sim-struct.v` files. The simulator harness uses a special pre-processor macro when instantiating the sorter (i.e., `EX_SORTER_IMPL` in `ex-sorter-sim-harness.v`). The simulator harness is responsible for handling command line arguments, instantiating the design, clocking the design, and of course setting up the specific input dataset we wish to use in our evaluation. A key difference between a simulator and a unit test, is that the simulator should also report various statistics that help us evaluate our design. The actual simulator files are relatively simple (i.e., `ex-sorter-sim-flat.v`, `ex-sorter-sim-struct.v`); they define the special implementation macro, include the implementation, and include the harness. We have already added the harness and simulator Verilog source files to the appropriate make variables in the subproject make fragment.

We can build and run a simulator using the following commands:

```bash
% cd ${TUTROOT}/build
% make ex-sorter-sim-flat
% ./ex-sorter-sim-flat +input=random +trace=1 +stats
```

The simulator should print out how many cycles it takes on average to sort four elements. Not surprisingly, it should take one cycle since we are using a fully pipelined implementation. The subproject’s makefile fragment includes special rules to run a selection of experiments using different implementations and input data sets. You can use the following to run all of these simulations and report the results:

```bash
% cd ${TUTROOT}/build
% make ex-sorter-eval
```
The output should confirm that the flat and structural implementations have the same performance on all input datasets. Note that if you have not actually completed the real implementation of the MinMax unit, the simulations will still run and actually they will also report what looks to be reasonable performance results; even though the structural implementation is not at all functionally correct. The take-away here is that you should not use the simulator and/or evaluation for verification; your testing strategy should be comprehensive enough that once you get to the evaluation you are confident that your design is fully functional.

6. Greatest Common Divisor: Verilog Design Example

In the final section of the tutorial, we will leverage what we learned in the previous sections to study a simple hardware unit that calculates the greatest common divisor (GCD) of two input operands. The code related to this part of the tutorial is located in the ex-gcd subproject. Figure 26 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 a latency-insensitive val/rdy interface on both the input and output. Our implementation will use Euclid’s algorithm to iteratively calculate the GCD. Figure 27 illustrates this algorithm as an executable Python function.

★ 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. Latency-Insensitive Message-Based Interfaces

The interface for the registered incremeniter in Section 4 included no extra control signals. A module which wants to use the registered incremeniter must explicitly handle the fact that the unit always takes exactly one cycle. The interface for the sorter in Section 5 included an extra valid signal. A module which wants to use the sorter could be carefully constructed so as to be agnostic to the latency of the sorter; this would enable flexibly trying out different sorting algorithms. One issue with including just a valid signal is that there is no way to know if the sorter is busy, and there is no way to tell the sorter that we are not ready to accept the result. In other words, there is no provision for back pressure. As shown in Figure 26, our GCD design will use a fully latency-insensitive interface by including two extra signals: a valid and a ready signal. These signals will allow additional flexibil-

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

Figure 26: 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.

Figure 27: 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.
ity: the GCD unit can indicate it is not ready to accept a new GCD input, and another module can indicate that it is not ready to accept the GCD output.

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

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

To complement the val/rdy microprotocol, we can use struct types to encapsulate the actual messages we send over a val/rdy interface. Take a look at the two struct types and corresponding unit test in `ex-gcd-msgs.v` and `ex-gcd-msgs.t.v`. We will use similar struct types for memory messages and network packets. We can use a unit test to verify that the message type is written correctly:

```
% cd ${TUTROOT}/build
% make ex-gcd-msgs-test && ./ex-gcd-msgs-test
```

★ To-Do On Your Own: Add a few more tests to the `ex_gcd_req_msg_t` test case in `ex-gcd-msgs.t.v` and verify that these values are correctly written into and read from the new struct type.

### 6.2. Functional-Level Modeling

Figure 26 illustrates how we might provide a functional-level model for our GCD unit. RTL models are synthesizable and meant to accurately model the target hardware, while a functional-level model is simply meant to accurately capture the input/output functional behavioral of the target hardware without any timing. A functional-level implementation of GCD is provided in `ex-gcd-GcdUnitFL.v` and a portion of this implementation is shown in Figure 28.

It should be clear from the code that this implementation is in no way synthesizable. Functional models often push most of the logic into a single large always @(posedge clk) concurrent block and make use of blocking assignments within this block to enable “procedural programming” similar to what you might use in a traditional imperative programming language (e.g., C or Python). The functional model actually uses Euclid’s algorithm to calculate the GCD, and this requires a data-dependent while loop which is also obviously not synthesizable. Since this is just a functional model, we are free to use any algorithm we wish as long as it has the correct input/output functional-level behavior.
// Implement GCD with Euclid's Algorithm

logic [\'EX_GCD_REQ_MSG_A_NBITS-1:0] A;
logic [\'EX_GCD_REQ_MSG_B_NBITS-1:0] B;
logic [\'EX_GCD_RESP_MSG_RESULT_NBITS-1:0] temp;
logic full, req_go, resp_go, done;

always @(posedge clk ) begin
    // Ensure that we clear the full bit if we are in reset.
    if ( reset )
        full = 0;
    // At the end of the cycle, we AND together the val/rdy bits to
    // determine if the input/output message transactions occurred.
    req_go = req_val && req_rdy;
    resp_go = resp_val && resp_rdy;
    // If the output transaction occurred, then clear the buffer full bit.
    // Note that we do this _first_ before we process the input
    // transaction so we can essentially pipeline this control logic.
    if ( resp_go )
        full = 0;
    // If the input transaction occurred, then write the input message
    // into our internal buffer and update the buffer full bit.
    if ( req_go ) begin
        A = req_msg.a;
        B = req_msg.b;
        full = 1;
    end
    // The output message is always the GCD of the buffer
    done = 0;
    while ( !done ) begin
        if ( A < B ) begin
            temp = A;
            A = B;
            B = temp;
        end
        else if ( B != 0 )
            A = A - B;
        else
            done = 1;
    end
    resp_msg.result <= A;
    // The output message is valid if the buffer is full
    resp_val <= full;
end
// Connect output ready signal to input to ensure pipeline behavior
assign req_rdy = resp_rdy;

Figure 28: Part of the Functional-Level Implementation GCD Unit – This code corresponds to the diagram in Figure 26 and implements the algorithm in Figure 27 using functional-level Verilog. This model is definitely not synthesizable.
Functional-level models will be critical for enabling the verification lead to start verification tasks early in the project. The verification lead can write and check tests using the functional-level model, and then gradually these same tests can be used with the RTL implementation. Using the functional-level model to write tests also ensures if the RTL implementation fails a test it is more likely due to the RTL implementation itself as opposed to an incorrect test case.

### 6.3. Verifying GCD Using Test Sources and Sinks

One of the nice features of using a latency-insensitive val/rdy interface is that it enables us to use a common framework for sending messages into the DUT and then verifying that the correct messages come out of the DUT. Our VC library includes `vc_TestRandDelaySource` and `vc_TestRandDelaySink` for this purpose, and we leverage these modules in building the unit tests for our functional-level GCD implementation. Take a closer look at the test harness in `ex-gcd-test-harness.v`. We instantiate a test source and attach it to the GCD unit’s input val/rdy interface, and then we instantiate a test sink and attach it to the GCD unit’s output val/rdy interface. Figure 29 illustrates the overall connectivity in the test harness. The test source includes the ability to randomly delay messages going into the DUT and the test sink includes the ability to randomly apply back-pressure to the DUT. By using various combinations of these random delays we can more robustly ensure that our flow-control logic is working correctly. Note that these unit tests illustrate both directed black-box and randomized black-box testing strategies. A similar approach of using Python-generated Verilog is used to create the random source/sink messages for randomized testing.

A common testing strategy is for the very first test-case to use directed source/sink messages with no random delays. For example, the first test case of our functional-level GCD implementation creates a couple of source messages and then manually we have determined the correct sink messages.

```bash
% cd ${TUTROOT}/build
% make ex-gcd-GcdUnitFL-test
% ./ex-gcd-GcdUnitFL-test +test-case=1 +trace=1
```

Recall that you can generate a VCD file to view waveforms using the `+dump-vcd` command-line option.

```bash
% cd ${TUTROOT}/build
% ./ex-gcd-GcdUnitFL-test +test-case=1 +dump-vcd
% gtkwave ex_gcd_GcdUnitFL-test.vcd
```

Once we know that our design works without any random delays, we continue to use directed source/sink messages but then add random source delays and random sink delays. For example, the second test case of our functional-level GCD implementation sets the test source to randomly

![Diagram](image.png)

**Figure 29: 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.
Figure 30: Line Trace for Functional-Level Implementation of GCD – . +indicates the val/rdy interface is not valid and not ready; # = val/rdy interface is valid but not ready; space = val/rdy interface is not valid and ready; message is shown when it is actually transferred across interface.

delay the input messages from zero to five cycles. We can also try using no delays on the source, but adding random delays to the sink, and finally add random delays to both the source and the sink. If we see that our design passes the tests with no random delays but fails with random delays this is a good indicator that there is an issue with our val/rdy logic.

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

Figure 30 illustrates a portion of the line trace for the randomized testing. Notice that the line trace tells something about what is going on with each val/rdy interface. A period (.) indicates that the interface is not ready but also not valid; a hash (#) indicates that the interface is valid but not ready; a space indicates that the interface is ready but not valid. The actual message is displayed when it is transferred from the producer to the consumer. We can see a message being sent into the GCD unit on cycle 19 and although the result is valid on cycle 20 the test sink is not ready until cycle 25 to accept the result. On cycles 20–21 the test source does not have a new message to send to the GCD unit. On cycle 22 it does indeed have a new message, but the GCD unit is not ready because it is still waiting on the test sink. Finally, on cycle 25 the test sink is ready and the GCD unit is able to send the result and accept a new input. The GCD unit takes a single cycle when there is no back pressure; we can see this on cycle 28–29.
To-Do On Your Own: Write a new task to generate source/sink messages inspired by the init_simple tasks currently in ex-gcd-test-harness.v. Your new task should be called init_coprime and it should include a few sets of relatively prime numbers. Two numbers are relatively prime (or coprime) if their greatest common divisor is one. Once you have defined this new task, create a new test case using the VC_TEST_CASE_BEGIN and VC_TEST_CASE_END macros. Your first new test case should set the random delays to zero, call your new init_coprime task, and then call the run_test task. Once this new test case is working, add more test cases that use the init_coprime task and change the random delays.

6.4. Control/Datapath Split Implementation

When modeling more complicated hardware, we will usually divide the design into two parts: the datapath and the control unit. The datapath contains the arithmetic operators, muxes, and registers that work on the data, while the control unit is responsible for controlling these components to achieve the desired functionality. The control unit sends control signals to the datapath and the datapath sends status signals back to the control unit. Figure 31 illustrates the datapath for the GCD unit and Figure 32 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 ex-gcd-GcdUnit.v.

Take a look at the datapath interface which is also shown in Figure 33. Notice how we use the GCD message types (i.e., ex_gcd_req_msg_t, ex_gcd_resp_msg_t) when specifying the input and output ports. We have encapsulated all of the control signals in their own struct type, and also all of the status signals in their own struct type. This approach enables compactly declaring the control/status signal ports in the datapath and control unit, and also enables compactly connecting the datapath and control unit together. Even more important, if we need to make a change or add a new control/status signal we only need to modify one place in our code instead of three.
Figure 33 also shows the first two datapath components, but take a look in `ex-gcd-GcdUnit.v` to see the entire datapath. Notice how we use a very structural implementation which exactly matches the datapath diagram in Figure 31. 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 34 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?).

The RTL implementation of the GCD unit uses the exact same test setup as for the functional-level implementation, even though the functional-level implementation always takes a single cycle while the RTL implementation takes a variable number of cycles; this illustrates the power of using latency-insensitive interfaces. Let’s run the unit tests for the GCD implementation.

```bash
% cd ${TUTROOT}/build
% make ex-gcd-GcdUnit-test && ./ex-gcd-GcdUnit-test +trace=1
```

Figure 35 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:** Write a new task to generate source/sink messages inspired by the `init_simple` tasks currently in `ex-gcd-test-harness.v`. Your new task should be called `init_same` and it should include a few sets of numbers where both operands are exactly the same. Once you have defined this new task, create a new test case using the `VC_TEST_CASE_BEGIN` and `VC_TEST_CASE_END` macros. Your new test case should set the random delays to zero, call your new `init_coprime` task, and then call the `run_test` task. Use the functional-level model to verify that your test case is correct before testing the RTL implementation. Use the line trace to understand how these kinds of inputs perform.
// Control signals (ctrl -> dpath)

typedef struct packed {
  logic a_reg_en; // Enable for A register
  logic b_reg_en; // Enable for B register
  logic [1:0] a_mux_sel; // Sel for mux in front of A reg
  logic b_mux_sel; // sel for mux in front of B reg
} ex_gcd_cs_t;

// Status signals (dpath -> ctrl)

typedef struct packed {
  logic is_b_zero; // Output of zero comparator
  logic is_a_lt_b; // Output of less-than comparator
} ex_gcd_ss_t;

// Datapath module

module ex_gcd_GcdUnitDpath
(
  input logic clk,
  input logic reset,

  // Data signals
  input ex_gcd_req_msg_t req_msg,
  output ex_gcd_resp_msg_t resp_msg,

  // Control and status signals
  input ex_gcd_cs_t cs,
  output ex_gcd_ss_t ss
);

localparam c_nbits = `EX_GCD_REQ_MSG_A_NBITS;

// A Mux

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

// A register

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

Figure 33: 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.
localparam a_x = 2'dx;
llocalparam a_ld = 2'd0;
llocalparam a_b = 2'd1;
llocalparam a_sub = 2'd2;
llocalparam b_x = 1'dx;
llocalparam b_ld = 1'd0;
llocalparam b_a = 1'd1;


task set_cs
  (input logic cs_req_rdy, input logic cs_resp_val, input logic [1:0] cs_a_mux_sel, input logic cs_a_reg_en, input logic cs_b_mux_sel, input logic cs_b_reg_en);
beg
req_rdy = cs_req_rdy;
resp_val = cs_resp_val;
cs.a_reg_en = cs_a_reg_en;
cs.b_reg_en = cs_b_reg_en;
cs.a_mux_sel = cs_a_mux_sel;
cs.b_mux_sel = cs_b_mux_sel;
end
endtask

logic do_swap;
logic do_sub;
assign do_swap = ss.is_a_lt_b;
assign do_sub = !ss.is_b_zero;

always @(*) begin
  set_cs( 0, 0, a_x, 0, b_x, 0 );
case ( state_reg )
  //
  // req resp a mux a b mux b
  //
  STATE_IDLE: set_cs( 1, 0, a_ld, 1, b_ld, 1 );
  STATE_CALC: if ( do_swap ) set_cs( 0, 0, a_b, 1, b_a, 1 );
  else if ( do_sub ) set_cs( 0, 0, a_sub, 1, b_x, 0 );
  STATE_DONE: set_cs( 0, 1, a_x, 0, b_x, 0 );
endcase
end

Figure 34: 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.
6.5. Exploring the GCD Implementation

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:

```bash
% cd ${TUTROOT}/build
% make ex-gcd-sim
% ./ex-gcd-sim +input=random-a +stats
% ./ex-gcd-sim +input=random-b +stats
```

You can run the entire evaluation with a single command like this:

```bash
% cd ${TUTROOT}/build
% make eval-ex-gcd
```

To-Do On Your Own: Optimize the GCD implementation to improve the performance on the two given input datasets. You should be able to save a few cycles by collapsing the IDLE and CALC states into a single state; allowing a direct transition from the CALC state back into the IDLE state when the output is ready by the input is not valid; and allowing a direct transition from the CALC state back into the CALC state when the output is ready and the input is valid. These optimizations should eliminate any bubbles and improve the performance of back-to-back GCD transactions. Another optimization approach 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.

Acknowledgments

This tutorial was developed for ECE 4750 Computer Architecture course at Cornell University by Christopher Batten.
# Appendix A: Constructs Allowed in Synthesizable RTL

<table>
<thead>
<tr>
<th>Always Allowed in Synthesizable RTL</th>
<th>Allowed in Synthesizable RTL With Limitations</th>
<th>Explicitly Not Allowed in Synthesizable RTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>logic</td>
<td>always(^1)</td>
<td>wire, reg(^{11})</td>
</tr>
<tr>
<td>logic [N-1:0]</td>
<td>enum(^2)</td>
<td>integer, real, time, realtime</td>
</tr>
<tr>
<td>&amp;</td>
<td>~</td>
<td>~ ~ (bitwise)</td>
</tr>
<tr>
<td>&amp; &amp;</td>
<td></td>
<td>!</td>
</tr>
<tr>
<td>&amp; ~ &amp;</td>
<td>~</td>
<td>~ ~ (reduction)</td>
</tr>
<tr>
<td>+ -</td>
<td>= (blocking assignment)(^6)</td>
<td>#N (delay statements)</td>
</tr>
<tr>
<td>&gt;&gt; &lt;&lt; &gt;&gt;&gt;</td>
<td>&lt;= (non-blocking assignment)(^7)</td>
<td>inout(^{14})</td>
</tr>
<tr>
<td>{}</td>
<td>typedef(^8)</td>
<td>initial</td>
</tr>
<tr>
<td>{}{} (repeat)</td>
<td>packed(^9)</td>
<td>variable initialization(^{15})</td>
</tr>
<tr>
<td>?:</td>
<td>$\text{signed}()(^{10})</td>
<td>negedge(^{16})</td>
</tr>
<tr>
<td>if else</td>
<td></td>
<td>casex, endcase</td>
</tr>
<tr>
<td>case, endcase</td>
<td></td>
<td>for, while, repeat, forever(^{17})</td>
</tr>
<tr>
<td>begin, end</td>
<td></td>
<td>fork, join</td>
</tr>
<tr>
<td>module, endmodule</td>
<td></td>
<td>deassign, force, release</td>
</tr>
<tr>
<td>input, output</td>
<td></td>
<td>specify, endspecify</td>
</tr>
<tr>
<td>assign</td>
<td></td>
<td>nmos, pmos, cmos</td>
</tr>
<tr>
<td>parameter</td>
<td></td>
<td>rmnos, rpmos, rcmos</td>
</tr>
<tr>
<td>localparam</td>
<td></td>
<td>tran, tranif0, tranif1</td>
</tr>
<tr>
<td>genvar</td>
<td></td>
<td>rtran, rtranif0, rtranif1</td>
</tr>
<tr>
<td>generate, endgenerate</td>
<td></td>
<td>supply0, supply1</td>
</tr>
<tr>
<td>generate for</td>
<td></td>
<td>strong0, strong1</td>
</tr>
<tr>
<td>generate if else</td>
<td></td>
<td>weak0, weak1</td>
</tr>
<tr>
<td>generate case</td>
<td></td>
<td>primitive, endprimitive</td>
</tr>
<tr>
<td>$\text{clog2}()</td>
<td></td>
<td>defparam</td>
</tr>
<tr>
<td>$\text{bits}()</td>
<td></td>
<td>unnamed port connections(^{18})</td>
</tr>
<tr>
<td>named port connections</td>
<td></td>
<td>unnamed parameter passing(^{19})</td>
</tr>
<tr>
<td>named parameter passing</td>
<td></td>
<td>all other keywords</td>
</tr>
<tr>
<td></td>
<td></td>
<td>all other system tasks</td>
</tr>
</tbody>
</table>

1. always 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 [$\text{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 @(•) blocks and are explicitly not allowed in always @(posedge clk) blocks.
Non-blocking assignments should only be used in `always @ (posedge clk)` blocks and are explicitly not allowed in `always @(*)` blocks.

$\text{signed}()$ can only be used around the operands to $\lll, \ggg, >, >=, <, <=$ to ensure that these operators perform the signed equivalents.

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

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

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

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

Multipliers can be synthesized and small multipliers can even be synthesized efficiently, but we do not allow this construct in the course. If you need to multiply or divide by a power of two, then you can use the left and right shift operators.

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

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.

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.

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

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

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.