ECE 429: Verilog Tutorial

Hiren Patel

May 10, 2020

Contents

List of Exercises

1 Introduction

This tutorial presents important Verilog constructs that will be useful in completing the project component for ECE 429. It is important to understand the difference between synthesizable and non-synthesizable register-transfer level (RTL) constructs. Although, we will not be validating our processor on a field-programmable gate array (FPGA), we will primarily use synthesizable RTL constructs to design the hardware. Some exceptions will be allowed, but they will be made explicitly clear within this tutorial.

1.1 Downloading the source

The source code, benchmarks, and examples for this class will be posted on `git.uwaterloo.ca`. Ensure that you have access to it as only Waterloo students will be able to access the repository. I will also use `git`, which means that you should become somewhat familiar with git. You will only have read-only permissions on the repository. The URL for the source code will be at `https://git.uwaterloo.ca/ece429`.

1.1.1 Distributing the source and tutorial

Please do not distribute the source code, benchmarks, or this tutorial. You do not have permission to distribute this, and or use this outside the purpose of this class.

1.1.2 Installing iverilog

Icarus Verilog (iverilog) is an open-source Verilog simulator. You can visit its site here: `http://iverilog.icarus.com/`. On Ubuntu, installing iverilog involves using `apt-get` to install iverilog. If you wish to install it from source, then please follow the installation instructions on iverilog's website.

Installing iverlog

```
sudo apt-get install iverilog
```

You can use Homebrew to install iverilog for MacOS. For Windows, you can get the binary from `http://bleyer.org/icarus/`.

1.1.3 Installing verilator

Verilator is an open-source Verilog simulator. It converts Verilog code into C++ or SystemC code, which is then compiled into native executable. Simulating large projects with verilator is much faster than with iverilog. However, there are some limitations in verilator. For example, delays in Verilog code are not allowed in verilator. You can visit its site here: `https://www.veripool.org/wiki/verilator`. On Ubuntu, installing verilator involves using `apt-get` to install verilator. If you wish to install it from source, then please follow the installation instructions on verilator's website.

Installing verilator

```
sudo apt-get install verilator
```

You can use Homebrew to install verilator for MacOS. Unfortunately for Windows users, you may need to use Cygwin or compile verilator from source using Microsoft Visual C++.

1.2 Simulating with iverilog

There are various simulation environments that one can use for the class project. My suggestions are to either use iverilog (`http://iverilog.icarus.com/`) or the student edition of ModelSim (`https://www.mentor.com/company/higher_ed/modelsim-student-edition`). The instructions in this tutorial will be specific to iverilog.

1.2.1 Compiling with iverilog

iverilog compiles the Verilog source files into an executable, which can then be run within your terminal. If you are interested in an integrated design environment (IDE) for iverilog, then you could consider using an Eclipse-based framework called IVI. My preference is to simply use emacs to write the Verilog specifications, and a terminal to compile them.

Let us consider an example where we wish to compile multiple `.v` files into an executable called `cpu`. You can execute the command below to do so. This will create an executable `cpu`, which you can then execute.

Compiling with iverilog

```
iverilog -g2005 -o cpu fetch.v decode.v execute.v memory.v writeback.v top.v
./cpu
```

You can certainly write a Makefile to help you compile Verilog code.

1.2.2 Installing gtkwave

Waveforms enable debugging of the hardware specification. gtkwave is an open-source value-change dump (VCD) file viewer. Once again, gtkwave can be installed on Ubuntu using `apt-get`.

Installing gtkwave

```
sudo apt-get install gtkwave
```

Please visit the website if you wish/need to compile from source.

1.2.3 Docker container

If you are familiar with docker, and wish to use a container for your course project, then you can do so. You can use the following Dockerfile to build your own docker image.

Docker image with iverilog, and gtkwave

```
FROM ubuntu:15.10
MAINTAINER rmrf@uwaterloo.ca

# Download and Install utilities
# ==============================
RUN rm /bin/sh && ln -s /bin/bash /bin/sh

RUN apt-get update \
  && apt-get install -y \
    vim \
    emacs \
    git

RUN apt-get install -y gcc python python-dev g++ build-essential zlib1g-dev

# Install iverilog & gtkwave
# ==========================
RUN apt-get install -y iverilog gtkwave
RUN apt-get install -y software-properties-common
RUN add-apt-repository -y ppa:saltmakrell/ppa
RUN apt-get update
RUN apt-get install -y git autotools-dev autoconf gperf
RUN apt-get install -y git flex bison

RUN bash -l -c "git clone git://github.com/steveicarus/iverilog.git; cd iverilog; sh
↪  autoconf.sh; ./configure; make; make install "
## cleanup

RUN apt-get clean && \
    cd /var/lib/apt/lists && rm -fr *Release* *Sources* *Packages* && \
    truncate -s 0 /var/log/*log

ENV PATH
↪  /usr/local/rvm/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

# Set environment variables.
ENV HOME /

WORKDIR /

# Simple test
#RUN iverilog -v

RUN /bin/bash -l -c "echo $'module counter( input x, output reg z); \n assign z = x +
↪  1; \n endmodule \n module test; initial begin \n \$display(\"ECE429: Test
↪  PASSED!\"); \n end \n endmodule' > /tmp/test.v; ls -l /tmp/*.v; cat /tmp/test.v;
↪  iverilog -v; iverilog -g2005 -o t /tmp/test.v; ./t; yosys -p \"hierarchy; proc;
↪  opt -full; stat;\" /tmp/test.v"
```

Please refer to docker documentation if you need further support.

1.2.4 Simulating and synthesizing a simple counter example

Listing 1 presents the Verilog specification for a counter in a crude hello world setting. At the moment, do not concern yourself with the different Verilog constructs as they will be discussed in further detail later in the tutorial. The purpose of this example is to show the necessary commands to compile the example, simulate it, show the output in a waveform, and synthesize it.

hello-world.v

```
module counter
  (
   input [3:0]  x,
   output [3:0] z
   );
   assign z = x + 1;
endmodule // counter

module top;
   reg [3:0] x;
   wire [3:0] z;
   initial begin
      // File for VCD.
      $dumpfile("hello-world.vcd");
      // All variables (0) from module
      ↪  instance c
      $dumpvars(0, c);
      #0 x = 1;
      #10 x = x + 1;
      #10 x = x + 1;
   end

   always @(*) begin
      $display("[Hello world] x:%h,
      ↪  z:%h",x,z);
   end
   counter c(.x(x),.z(z));
endmodule // top
```

Listing 1: Hello world with a counter.

1. Compile the example. We are using the 2005 Verilog standards using the `-g` flag.

   Hello World with AND gate

   ```
   iverilog -g 2005 -o hello-world hello-world.v
   ```

2. You should have an executable called `hello-world`. Run the executable.

   Simulate example.

   ```
   ./hello-world
   ```

3. The following output should appear on your terminal.

   Output of simulation.

   ```
   VCD info: dumpfile hello-world.vcd opened for output.
   [Hello world] x:1, z:2
   [Hello world] x:2, z:3
   [Hello world] x:3, z:4
   ```

4. There should be a `hello-world.vcd` file there too, which you can open using gtkwave. Once it is open, you can click on `top`, followed by `c`, and then add the `x` and `z` wires using the `insert` button.

   Open gtkwave

   ```
   gtkwave hello-world.vcd
   ```

5. Below is a screenshot of what the waveform should look like.

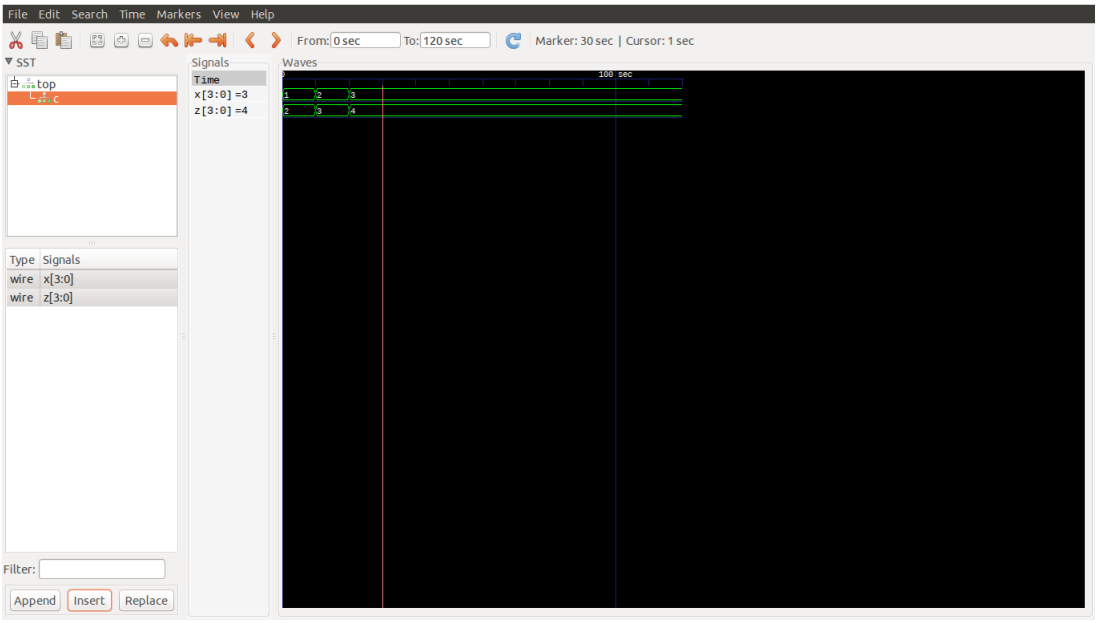

Figure 1: Screenshot of gtkwave.

When working on a large project, automating the process of adding and manipulating signals using a tcl script is important because working on gtkwave GUI would be ineffective. The following simple tcl script shows how to add signals and comments on the gtkwave window and then change the color of each signals.

Below is a screenshot of what the waveform should look like with the tcl script.

Open gtkwave

```
gtkwave -f hello-world.vcd -S hello-world.tcl
```

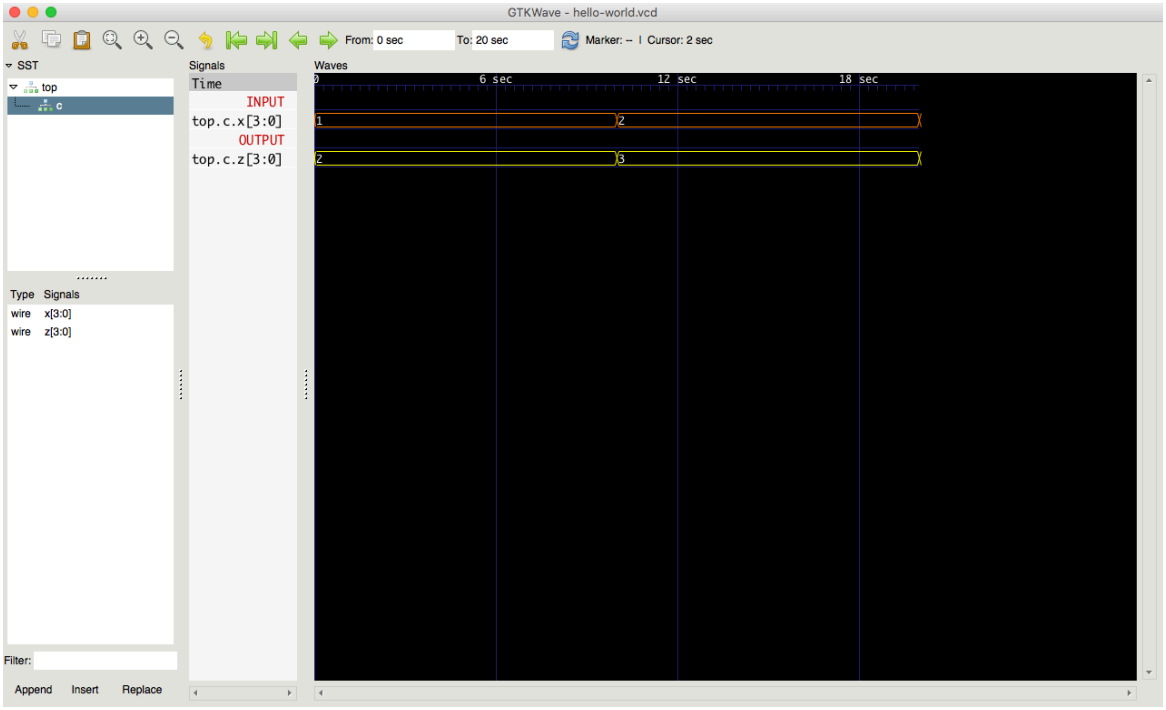

Figure 2: Screenshot of gtkwave with tcl script.

hello-world.tcl

```
#!/usr/bin/tclsh
#
# simple example of using tcl with gtkwave:


# Query the dumpfile for signals with "clk" or [1:48] in the signal name
set nfacs [gtkwave::getNumFacs]
set dumpname [ gtkwave::getDumpFileName ]
set dmt [ gtkwave::getDumpType ]

puts "number of signals in dumpfile '$dumpname' of type $dmt: $nfacs"

set clk48 [list]
#puts $clk48

# Show input and output signals
lappend clk48 "top.c.x\[3:0\]"
lappend clk48 "top.c.z\[3:0\]"
set ll [ llength $clk48 ]

puts "number of signals found matching either 'clk' or '\[1:48\]': $ll"

# Add "INPUT" comment first
gtkwave::/Edit/Insert_Comment "INPUT"

# Add top.c.x and top.c.x signals
set num_added [ gtkwave::addSignalsFromList $clk48 ]
puts "num signals added: $num_added"

# Change color of singnal top.c.x to Orange
gtkwave::/Edit/Highlight_Regexp "x"
gtkwave::/Edit/Color_Format/Orange
gtkwave::/Edit/UnHighlight_All

# Add "OUTPUT" comment above top.c.z
gtkwave::/Edit/Highlight_Regexp "x"
gtkwave::/Edit/Insert_Comment "OUTPUT"
gtkwave::/Edit/UnHighlight_All

# Change color of singnal top.c.x to Yellow
gtkwave::/Edit/Highlight_Regexp "z"
gtkwave::/Edit/Color_Format/Yellow
gtkwave::/Edit/UnHighlight_All

gtkwave::/View/Show_Wave_Highlight
gtkwave::/Edit/Set_Trace_Max_Hier 0

# zoom full
gtkwave::/Time/Zoom/Zoom_Full
```

Listing 2: Hello world tcl script.

1.3 Simulating with verilator

Verilator provides a much faster simulation speed compared to iverilog. For detailed usage of verilator, please refer to the documentation of verilator (https://www.veripool.org/projects/verilator/wiki/Manual-verilator).

1.3.1 Compiling with verilator

Compiling with verilator involes two steps. In the first step, verilator converts Verilog code into C++ or SystemC code, with necessary assistant files. In the second step, the C++ compiler compiles the generated code into binary executable. If we want to combine multiple `.v` files into an executable, and the top module is named `top`, we can use the following command to generate a binary executable `Vtop`.

Compiling with verilator

```
verilator -sv --top-module top --cc fetch.v decode.v execute.v memory.v writeback.v
↪  top.v --exe tb.cpp # step 1
cd obj_dir
make -j -f Vtop.mk Vtop #step 2
./Vtop
```

You will need make to compile the generated C++ code.

1.3.2 Simulating a simple counter example

We will use the Verilog code in listing 3 to demonstrate how to use verilator to simulate the code.

hello-world-2.v

```
1  module counter
2    (
3     input [3:0]  x,
4     output [3:0] z
5     );
6     assign z = x + 1;
7  endmodule // counter
```

Listing 3: Hello world with a counter, verilator version.

Apart from that, we also need a C++ file to act as the testbench, as is shown in listing 4.

hello-world-2.cpp

```cpp
#include <cstdio>
#include <cstdlib>

#include <verilated.h>

#include "Vcounter.h"

using namespace std;

Vcounter *top;

vluint64_t main_time = 0; // Current simulation time

double sc_time_stamp () { // Called by $time in Verilog
  return main_time;
}


int main(int argc, char** argv) {
  Verilated::commandArgs(argc, argv); // Remember args

  top = new Vcounter; // Create instance

  top->x = 1;
  top->eval();
  printf("[Hello World] x:%x, z: %x\n", top->x, top->z);

  main_time += 10;
  top->x = top->x + 1;
  top->eval();
  printf("[Hello World] x:%x, z: %x\n", top->x, top->z);

  main_time += 10;
  top->x = top->x + 1;
  top->eval();
  printf("[Hello World] x:%x, z: %x\n", top->x, top->z);

  top->final();               // Done simulating
  delete top;

  return 0;
}
```

Listing 4: Hello world with a counter.

1. Convert the example into C++ files. This will generate an `obj_dir` folder, which includes assistant files.

   Hello World with verilator

   ```
   verilator -sv --cc hello-world-2.v --top-module counter --exe hello-world-2.cpp
   ```

2. Change directory to `obj_dir` and compile the generated C++ sources.

   Compile the example.

   ```
   cd obj_dir
   make -j -f Vcounter.mk Vcounter
   ```

3. You should have an executable called `Vcounter`. Run the executable.

   Simulate the example.

   ```
   ./Vcounter
   ```

4. The following output should appear on your terminal.

   Output of simulation.

   ```
   [Hello World] x:1, z: 2
   [Hello World] x:2, z: 3
   [Hello World] x:3, z: 4
   ```

1.3.3 Generating waveform with verilator

It is possible to generate waveform file (`.vcd`) with verilator. We demonstrate the steps for generating waveform file with the Verilog code in listing 3. We need to change the C++ testbench from listing 4 to listing 5.

hello-world-2-vcd.cpp

```
#include <cstdio>
#include <cstdlib>

#include <verilated.h>

#include "Vcounter.h"
#include "verilated_vcd_c.h"

using namespace std;

Vcounter *top;

vluint64_t main_time = 0; // Current simulation time

double sc_time_stamp () { // Called by $time in Verilog
  return main_time;
}


int main(int argc, char** argv) {
  Verilated::commandArgs(argc, argv); // Remember args

  top = new Vcounter; // Create instance

  Verilated::traceEverOn(true);
  VerilatedVcdC* tfp = new VerilatedVcdC;
  top->trace (tfp, 99);
  tfp->open ("hello-world-2.vcd");

  top->x = 1;
  top->eval();
  printf("[Hello World] x:%x, z: %x\n", top->x, top->z);
  tfp->dump (main_time);

  main_time += 10;
  top->x = top->x + 1;
  top->eval();
  printf("[Hello World] x:%x, z: %x\n", top->x, top->z);
  tfp->dump (main_time);

  main_time += 10;
  top->x = top->x + 1;
  top->eval();
  printf("[Hello World] x:%x, z: %x\n", top->x, top->z);
  tfp->dump (main_time);

  tfp->close();
  top->final();               // Done simulating
  delete top;

  return 0;
}
```

Listing 5: Hello world with a counter with vcd dump.

1. Convert the example into C++ files. Now we add `--trace` option.

   Hello World with verilator

   ```
   verilator --trace -sv --cc hello-world-2.v --top-module counter --exe hello-world-2-vcd.cpp
   ```

2. Change directory to `obj_dir` and compile the generated C++ sources.

   Compile the example.

   ```
   cd obj_dir
   make -j -f Vcounter.mk Vcounter
   ```

3. You should have an executable called `Vcounter`. Run the executable.

   Simulate the example.

   ```
   ./Vcounter
   ```

4. The following output should appear on your terminal.

Output of simulation.

```
[Hello World] x:1, z: 2
[Hello World] x:2, z: 3
[Hello World] x:3, z: 4
```

5. You should also see a `hello-world-2.vcd` waveform in the same directory, which is similar to the one in subsubsection 1.2.4

Output of simulation.

```
$ ls .
Vcounter
...
hello-world-2.vcd
```

2 Verilog

2.1 Basic register versus wires

There are two basic data types in Verilog: registers and wires. The simplest way to differentiate the two is to consider them as hardware structures. Wires, denoted as **wire**, are best considered as physical entities that propagate a value across it. Registers denoted as **reg**, on the other hand, can represent sequential elements (hardware components that hold state), and **confusingly** also physical entities that propagate values (wires). The manner in which a register is used distinguishes between whether the specification is a wire or a sequential register element. For now, we will consider **reg** as a way to declare variables, and later discover how to synthesize wires and/or sequential elements.

2.2 Bit basics

Table 1 shows the some of the common Verilog operators that we will employ in this class. While most of these operators are synthesizable, note that `*`, `/` and `%` are typically not implemented using automatic synthesis. For instance, recall from ECE222 that there are different ways of designing a multiplier: sequential or using Booth's method. In this class, we will not be implementing efficient versions of multipliers or divisors. Therefore, we are allowed to use `*` and `/` only in the ALU.

Bitwise Operators

Operator	Name
~	Bitwise Negation
&	Bitwise AND
|	Bitwise Inclusive OR
^	Bitwise Exclusive OR

Logical Operators

Operator	Name
!	Logical Negation
&&	Logical AND
||	Local OR

Arithmetic Operators

Operator	Name
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus

Reduction Operators

Operator	Name
&	Reduction AND
~&	Reduction NAND
|	Reduction OR
~|	Reduction NOR
^	Reduction XOR
~^	Reduction XNOR

Relational Operators

Operator	Name
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
==	Equal
!=	Not equal

Shift Operators

Operator	Name
<<	Shift left
>>	Shift right
>>>	Arithmetic shift right
<<<	Arithmetic shift left

Assignment Operators

Operator	Name
=	Blocking
<=	Non-blocking

Other Operators

Operator	Name
{}	Concatenate
{N{}}	Replicate N times

Table 1: Verilog operators.

Bit management and manipulation is necessary in hardware design. The Verilog specification in Listing 6 provides simple examples of declaring 1-bit and 4-bit registers, and simple operations on them. I will refer to a vector of bits (1-bit, 4-bit, n-bit) as bit strings, and selecting a portion of these bit strings as substrings. For the most part, this example is self-explanatory, but I will attempt to identify a few important syntactical observations below.

- Declaring an n-bit register requires specifying the bit-width: `reg [3:0] x2, y2, z2;`
- Assigning values to a register requires specifying the width of the assigned value. For example, `x = 1'b0;` denotes that `x` gets assigned a 1-bit binary (`b`) value of `0`. Similarly, `x2 = 4'h3;` shows an example of assigning a 4-bit hexadecimal (`h`) value of `3` to `x2`.
- A non-synthesizable construct `$display(...)` is equivalent to the infamous `printf(...)`. The syntax is similar as well, and self-explanatory from the examples.
- Any Verilog specification that is within an **initial** block is non-synthesizable. The **initial** block is executed before any of the processes are run. A caveat to be aware of is that yosys does do some clever synthesis passes of which one includes using specification within the **initial** block for synthesis. Hence, please be careful with the synthesis results, and your specification to confirm that your specification yields the hardware you desire.

Listing 6 also shows commonly used logical, and Boolean operators on both the 1-bit and 4-bit registers.

basic-bits.v

```
module basic_bits;
   // 1 bit registers
   reg x,y,z;
   // 4 bit registers
   reg [3:0] x2, y2, z2;

   initial begin
      // Bit type includes: b = {0,1,X,Z}
      x = 1'bX;  y = 1'bZ;    $display("1'b0: %b, 1'b1: %b ", x, y);
      x = 1'b0;  y = 1'b1;    $display("1'b0: %b, 1'b1: %b ", x, y);
      // Hex type includes: h={0,1,2,...,A,B,..,F}
      // Decimal time includes: d={...}
      x2 = 4'h3;  y2 = 4'hB;     $display("4'h3: %h, 4'hB: %h ", x2, y2);
      x2 = 4'd3;  y2 = 4'd11;    $display("4'd3: %d, 4'd11: %d ", x2, y2);
      // Logical operators: AND, OR, XOR, NOT, NAND
      z = x & y;   $display("z = x & y :=> %b = %b & %b ", z, x, y);
      z = x | y;   $display("z = x | y :=> %b = %b | %b ", z, x, y);
      z = x ^ y;   $display("z =x ^ y  :=> %b = %b ^ %b ", z, x, y);
      z = ~ (x & y);   $display("z =~(x & y)  :=> %b = ~(%b & %b) ", z, x, y);
      z = ~x;    $display("z = ~x    :=> %b = ~ %b ", z, x);
      // Boolean logical operators: &&, ||, !
      z = x && y;  $display("z = x && y :=> %b = %b && %b ", z, x, y);
      z = x || y;  $display("z = x || y :=> %b = %b || %b ", z, x, y);
      z = !x;    $display("z = !x    :=> %b = ! %b ", z, x);
      // 4 bit register operations
      // Logical operators: AND, OR, XOR, NOT, NAND
      z2 = x2 & y2;   $display("z2 = x2 & y2 :=> %b = %b & %b ", z2, x2, y2);
      z2 = x2 | y2;   $display("z2 = x2 | y2 :=> %b = %b | %b ", z2, x2, y2);
      z2 = x2 ^ y2;   $display("z2 =x2 ^ y2  :=> %b = %b ^ %b ", z2, x2, y2);
      z2 = ~ (x2 & y2); $display("z2 =~(x2 & y2)  :=> %b = ~(%b & %b) ", z2, x2, y2);
      z2 = ~x2;    $display("z2 = ~x2    :=> %b = ~ %b ", z2, x2);
      // Boolean logical operators: &&, ||, !
      z2 = x2 && y2;  $display("z = x && y :=> %b = %b && %b ", z2, x2, y2);
      z2 = x2 || y2;  $display("z = x || y :=> %b = %b || %b ", z2, x2, y2);
      z2 = !x2;    $display("z = !x    :=> %b = ! %b ", z2, x2);
   end // initial begin
endmodule // basic_bits
```

Listing 6: Basic bit manipulation.

2.3 Shifting bits

Understanding bit manipulation is important to effectively implement hardware using RTL. Shifting bits is one of these important mechanisms. Bit shifts can either happen in the left or right direction. Recall that there are two variants of shifts: logical and arithmetic. A logical shift inserts a `0` at either end of the bit string to shift the bit string. An arithmetic shift left is equivalent to a logical shift left; however, an arithmetic shift right, inserts the sign bit at the most significant bit. Note that the key distinguishing fact between the arithmetic shift right and logical shift right is that the arithmetic shift preserves the sign of the shifted value.

- Be default, values are assumed to be unsigned. Therefore, to ensure correct sign representation, one must use `$signed()` to ensure that the value is interpreted as a signed value.
- You should avoid using `$signed()` declarations for data types. I would suggest you simply use unsigned data types, and perform these explicit casts using `$signed()` whenever you want to ensure correct sign representation.
- Notice in the examples for arithmetic shift right, I explicitly cast `x` to be signed.

shift-bits.v

```
module shift_bits;
   // 8-bit registers
   reg [7:0] x, y, z;

   initial begin
      x = 8'hB7;  $display("x = %b", x);

      // Logical shift left
      z = x << 1;  $display("x:8'b1011_0111 << 1 :=> x = %b", z);
      z = x << 2;  $display("x:8'b1011_0111 << 2 :=> x = %b", z);
      z = x << 3;  $display("x:8'b1011_0111 << 3 :=> x = %b", z);
      z = x << 4;  $display("x:8'b1011_0111 << 4 :=> x = %b", z);
      // Logical shift right
      $display("logical shift left");
      z = x >> 1;  $display("x:8'b1011_0111 >> 1 :=> x = %b", z);
      z = x >> 2;  $display("x:8'b1011_0111 >> 2 :=> x = %b", z);
      z = x >> 3;  $display("x:8'b1011_0111 >> 3 :=> x = %b", z);
      z = x >> 4;  $display("x:8'b1011_0111 >> 4 :=> x = %b", z);
      // Signed arithmetic shift right
      $display("signed arithmetic shift right");

      // Selective most-significant bit assignment
      x[7] = 1'b1;

      // Notice the most-significant bit is being replicated!
      z = $signed(x) >>> 1;  $display("x:8'b1011_0111 >>> 1 :=> x = %b", z);
      z = $signed(x) >>> 2;  $display("x:8'b1011_0111 >>> 2 :=> x = %b", z);
      z = $signed(x) >>> 3;  $display("x:8'b1011_0111 >>> 3 :=> x = %b", z);
      z = $signed(x) >>> 4;  $display("x:8'b1011_0111 >>> 4 :=> x = %b", z);
   end // initial begin
endmodule // shift_bits
```

Listing 7: Shifting bits.

2.4 Arithmetic operations

Arithmetic operations consist of the regular addition, subtraction, multiplication, division, and modulus operators. Listing 8 shows examples illustrating the use of each of these. While we are going to use all the above mentioned arithmetic operations, please understand that automatic synthesis of complex arithmetic components such as multipliers and divisors are largely inefficient. Consequently, in production designs these arithmetic units are designed separately (recall the different ways in which one can build hardware units for multiplication and division from ECE 222). They are then structurally connected.

arith-bits.v

```
1  module arith_bits;
2     // 8-bit registers
3     reg [7:0] x, y, z;
4
5     initial begin
6        x = 8'hB7;  $display("x = %h", x);
7        y = 8'h11;  $display("y = %h", y);
8
9        z = x + y;  $display("z = %h", z);
10       z = x - y;  $display("z = %h", z);
11       z = x[3:0] * y[3:0];  $display("z = %h", z);
12
13       y = 8'h02;
14       z = x / y;  $display("z = %h", z);
15    end // initial begin
16 endmodule // arith_bits
```

Listing 8: Arithmetic operations.

- You will notice in Listing 8 that when using `*` we are using another Verilog operator to select a 4-bit substring from each of the `x` and `y` registers. The bit selection operator allows us to select a consecutive substring of bits from a register or wire. This is necessary for a multiplication because the product of two n-bit strings provides a 2n result.

2.5 Concatenation operation

Concatenation enables the merging of multiple bit strings together. Replication does what the name suggests: replicates the bit a given number of times. The examples in Listing 9 provide simple, but self-explanatory examples.

concat-bits.v

```
1  module concat_bits;
2     // 8-bit registers
3     reg [7:0] x, y, z;
4     initial begin
5        x = 8'hB7;  $display("x = %b", x);
6        y = 8'h11;  $display("y = %b", y);
7        // Signed arithmetic shift right
8        $display("Concats");
9        z = {x[3:0],y[3:0]};
10       $display("x[3:0] = %b, y[3:0] = %4b, z = %b",
11                x[3:0], y[3:0], z);
12       // Replication used for 2 least-significant bits
13       z = {x[2:0],y[2:0], {2{1'b1}} };
14       $display("x[2:0] = %b, y[2:0] = %4b, z = %b",
15 x[2:0], y[2:0], z);
16    end // initial begin
17 endmodule // concat_bits
```

Listing 9: Concatenation and replication.

3 Common Hardware Structures

I will attempt to introduce common hardware structures and Verilog specifications that can synthesize them. Through this exposition, I will continue to introduce important Verilog constructs, and offer rule-of-thumb design guidelines.

3.1 Decoders

Decoders play an important role in designing processors. They are used to identify the instruction to be executed. This requires inspecting different portions of the bit string to determine the type of the instruction, its operands, and any other particular information relevant for the processing of the instruction.

The recommended way to design decoders is using a Verilog **case** statement. Let us implement a simple 3:8 decoder as shown in Listing 10. A 3:8 decoder takes a 3-bit input, and generates an 8-bit select signal. We take this example further, and build the decoder as a combinational component. This means that there are no state elements within the circuit.

These are the following important observations from this example.

- One can certainly use **if** constructs to implement the respective decoding logic. However, I recommend using **case** statements whenever the decoding involves more than one case. This is because multiple **if** statements can become difficult to decipher, and for one to be able to to synthesize to hardware. This is because for synthesis, one must ensure that all fall-through paths are specified.
- When you expect the decoding to be complex, I recommend that `define`s are used effectively. The example uses it to simply illustrate how they can be used. You would certainly want to use this for constants, and instruction encodings when you write your own decoder for the project. This will certainly help you in building a correct processor, and ease in debugging.
- Always have a **default** case. This acts as a case to catch any unexpected oversights or incompletely specified **case** statements. This will also ensure that your hardware would synthesize. Note that `$display()` is not synthesizable.
- The **module** declaration now specifies the parameters of `x, select`, and whether they are inputs or outputs. `x` is an input that is a 3-bit wide bit string. When no datatype (**wire** or **reg**) is specified, then by default the datatype is a **wire**. Notice that `select` is specified as an **output reg**. This indicates that `select` is an output, and it is a registered output. As you progress through the tutorial, you will understand that any assignments that happen within an **always** block must have left-hand side variables as registers. Otherwise, the Verilog specification is syntactically incorrect.
- The **always** block within which the decoder is implemented has a sensitivity list with `x` in it. This means that whenever there is a change on the input `x` the Verilog specification within the **always** block gets scheduled to execute. This should remind you have process blocks in VHDL, and their sensitivity lists.
- The `decoder` module gets instantiated in the `dut` module with an instance name `d`. Notice how the registers, and wires from the `dut` are connected to the inputs and outputs of the decoder instance. `.a(b)` denotes that `b` is bound to the input/output `a`. Module instances are declared outside **initial**, and **always** blocks.
- The **initial** block drives the inputs as before. At the beginning of simulation with time event #`0`, the value of `x` and `clock` are set. Notice the `$finish`, which notifies the simulator to finish the simulation at a timed event of calculated as a #`100` units away from the current time stamp.
- The **always** `@(`**posedge** `clock)` statement has the `clock` in the sensitivity list. Further, the **posedge** indicates that the process (code within the **always**) `block` is scheduled to execute only when there is a positive edge of the `clock` (rising edge). This is done to change the input values `x`, and to print the output to the screen.

decoder.v

```
`define SIX   6
`define SEVEN 7

module decoder (x, select );
   input [2:0] x;
   output reg [7:0] select;
   always @(x) begin
      case (x)
        3'b000: begin
           select = 8'b0000_0001;
        end
        // Single line assignment
        3'b001:  select = 8'b0000_0010;
        3'b010:  select = 8'b0000_0100;
        3'b011:  select = 8'b0000_1000;
        3'b100:  select = 8'b0001_0000;
        3'b101:  select = 8'b0010_0000;
        // Using defines
        `SIX:    select = 8'b0100_0000;
        `SEVEN:   select = 8'b1000_0000;
        // Always have a default to catch unexpected oversights
        default: begin
          $display("ERROR");
           select = 8'h0;
        end
      endcase // case (x)
   end // always @ (x)
endmodule // decoder

module dut;
   reg clock;   reg [2:0] x;   wire [7:0] select;

   initial begin
      #0 x = 3'h0; clock = 0;
      #100 $finish;
   end
   // Toggle clock signal every 5
   always begin
      #5 clock = ~clock;
   end
   always @(posedge clock) begin
      x = x + 1;
      $display("time: %t, x: %b, select: %b", $time,x,select);
   end
   // Instantiate module
   decoder d( .x(x),
              .select(select)
              );
endmodule // dut
```

Listing 10: 3:8 Decoder.

3.2 Multiplexors

Multiplexors provide a way to select a subset of outputs from a larger number of inputs. A simple example of when a multiplexor (mux, for short) is necessary is when there are two or more **`wire`**s that may be connected to a single **`wire`** depending on a certain condition. Clearly, two input wires cannot drive a single output wire; thus, there must be exclusive connection of one input wire to the output at any given time instance. A mux enables this.

Listing 11 shows an example of a 2:1 mux. This mux has `x, y, sel` as inputs, `z` as output. `sel` enables selecting which of the `x, y` inputs are selected for the output `z` using a continuous statement. The continuous statement that **`assign`**s the result to `z` uses a ternary operator. The ternary operator suggest that if `sel` is `1` then connect `x` to the output `z`; otherwise, connect `y` to the output `z`. Note that one could have used a **`case`** statement to achieve the same hardware. However, ternary operators are compact representations, and often preferred for such situations. In addition, any combinational circuit that can be specified using continuous assignment is typically preferred.

mux.v

```
1  module mux_2_to_1 ( x, y, z, sel );
2     input wire x, y, sel;
3     output wire z;
4     // Ternary operator
5     assign z = (sel) ? x : y;
6  endmodule // mux
7
8  module dut;
9     reg x, y, sel;
10    wire z;
11
12    initial begin
13       #0  x = 0; y = 1; sel=0;
14       #5  x = 1; y = 1; sel=0;
15       #10 x = 1; y = 0; sel=0;
16       #15 x = 0; y = 0; sel=0;
17       #20 x = 0; y = 1; sel=1;
18       #25 x = 1; y = 1; sel=1;
19       #30 x = 1; y = 0; sel=1;
20       #35 x = 0; y = 0; sel=1;
21    end
22    // Instantiate MUX gate
23    mux_2_to_1 gate ( .x(x), .y(y), .z(z),
      ↪  .sel(sel) );
24
25    always @(*) begin
26       $display("time = %t, x = %b, y =
         ↪  %b, sel = %b, z = %b",
27                $time, x, y, sel, z);
28    end
29 endmodule // dut
```

Listing 11: Mux gate.

3.3 Combinational or sequential

It is important to distinguish between specifications that will synthesize to combinational or sequential hardware. To illustrate the ways in which Verilog allows one to do this, we will walk through a few simple examples.

3.3.1 Assigns for combinational circuits

We will start with an example of an AND gate circuit to illustrate the different approaches in Verilog to describe combinational circuits as shown in Listing 12. Note the following.

- The `and_assign` module has two inputs `x, y`, and one output `z`. Without specifying the bit-width, the widths default to 1-bit bit strings.
- The continuous assignment statement is done using the **`assign`** construct, which performs a bitwise AND on `x,y`, and assigns the output to `z`. Assigns are synthesized as combinational circuits.

We are also introducing a testbench in the example in Listing 12. Notice that module `dut` is our testbench that tests the design under test (dut). Testbenchs are not synthesizable.

- Notice that we introduce logic to toggle the clock. This is only necessary for this example to increment the input `x` so that we can see different decoded values on `select`.
- We use **`reg`** for `clock, x`, and **`wire`** for `select`. A simple rule-of-thumb is that whenever there is an assignment within an **`always`** block, the left-hand side variable must be a register. Therefore, you cannot assign a **`wire`** variable within an **`always`** block. You can, however, assign a **`wire`** as continuous statements as shown in Listing 12.
- For the **`always`** `@(z)` statement, `z` is in the sensitivity list for the particular process. This means that whenever the value of `z` changes, the Verilog specification within the **`always`** block is scheduled to execute.
- The statements preceded by a `#5` or `#10` indicate timed events. Recall from ECE327 that the simulation of Verilog uses the discrete-event semantics. Hence, these timed events effectively schedule the statements that follow the timed event to occur after the specified time. Consequently, the values of `x,y` change at those time instances.
- Instantiating a module requires port to wire binding. In this example, we instantiate a `gate` of type `and_assign`, and bind the ports `x, y, z` with their respective wires.

and-assign.v

```
1  module and_assign ( x, y, z );
2     input wire x, y;
3     output wire z;
4     // Continuous assignment
5     assign z = x & y;
6  endmodule // and_assign
7
8  module dut;
9     reg x, y;
10    wire z;
11
12    initial begin
13       #0  x = 0; y = 1;
14       #5  x = 1; y = 1;
15       #10 x = 1; y = 0;
16       #15 x = 0; y = 0;
17    end
18    // Instantiate AND gate
19    and_assign gate ( .x(x), .y(y),
      ↪  .z(z));
20
21    always @(z) begin
22       $display("time = %t, x = %b, y =
         ↪  %b, z = %b", $time,x,y,z);
23    end
24 endmodule // dut
```

Listing 12: AND Gate: Continuous assignment.

3.3.2 Always for combinational circuits

We change the example using **`assign`** to use **`always`** constructs to describe combinational circuits. We make minor changes to the example from Listing 12 to give us Listing 13. The main changes are as follows.

- We change the declaration of `z` to be of type **`output reg`**. Recall from before that left-hand-side variables cannot be **`wire`**s within an **`always`** block.

- An `always @(*) begin` statement is used to wrap the assignment to `z` with the result of the AND between `x,y`. Note that the `@(*)` is a short form to denote that all right-hand side wires are in the sensitivity list. Alternatively, one could have specified it as `@(x,y)`, and it would be equivalent for this example. This means that whenever `x,y` changes, output for `z` would be computed. This represents the way combinational circuits work: whenever the inputs change, the signals propagate to the output. In this case, the propagation happens to go through the AND operation.
- This example shows the **confusing** aspect of using registers to specify combinational circuits. In general, simple combinational circuits such as the one in this example should use the `assign` construct. This reduces any ambiguity in the specification. However, for complex combinational logic, one may want to use `always` constructs. In this case, the rule-of-thumb to follow is that **one must only use blocking assignments (using =) when the `always @(*)` statements is used.**

and-always-comb.v

```
1  module and_always_comb ( x, y, z );
2     input x, y;
3     output reg z;
4     // Always block for combinational
      ↪  circuit.
5     always @(*) begin
6        z = x & y;
7     end
8  endmodule // and_assign
9
10 module dut;
11    reg x, y;
12    wire z;
13
14    initial begin
15       #0  x = 0; y = 1;
16       #5  x = 1; y = 1;
17       #10 x = 1; y = 0;
18       #15 x = 0; y = 0;
19    end
20    // Instantiate AND gate
21    and_always_comb gate ( .x(x), .y(y),
      ↪  .z(z));
22
23    always @(z) begin
24       $display("time = %t, x = %b, y =
         ↪  %b, z = %b", $time,x,y,z);
25    end
26 endmodule // dut
```

Listing 13: AND Gate: Combinational using `always`.

3.4 Sequential components

Sequential components enable holding state for one or more clock cycles. You may recall this from ECE124, ECE222, and ECE327. Flip-fops and registers are examples of sequential components. In order to design sequential components, we must understand the difference between **blocking** and **non-blocking** assignments.

3.4.1 Blocking and Non-blocking Assignments

Intuitively, a **blocking** assignment "blocks" at the blocking assignment until the assignment is complete before proceeding to the next assignment. In other words, the evaluation and assignment are performed immediately. An = operator is used for **blocking** assignment. Most of the examples we have seen until now have used this **blocking** assignment. Note that this is synonymous to variables in languages such as C/C++. On the contrary, **non-blocking** assignments behave differently, and are denoted by the <= assignment operator. The **non-blocking** assignment defers all assignments until all right-hand sides have been evaluated at the end of the simulation step.

Pipeline example The best way to distinguish non-blocking and blocking assignments is through an example. In ECE429, you will be designing a pipelined processor; hence, knowing how to use Verilog to create pipelines is essential. To illustrate **non-blocking** versus **blocking**, we will design a 3-stage pipeline of incrementers as shown in Figure 3.

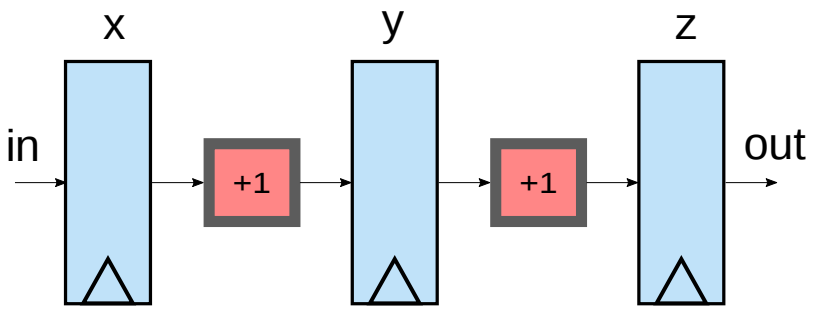

Figure 3: 3-stage pipeline of incrementers.

Listing 14 shows an implementation of the `non-blocking` module that implements the pipeline shown in Figure 3, and the terminal output of the simulation (below the dotted line).

- Note that the **`always`** block has a sensitivity list of `@(posedge clock)`. This indicates that the specification within the **`always`** block gets scheduled to execute whenever there is a positive edge of the `clock`.
- Every assignment within the **`always`** block uses the <= non-blocking assignment operator. This means that whenever there is a positive edge of the `clock`, the values of `in, x, y` are read in a non-determined order, and once they are stable (values through combinational circuits that drive `in, x, y` have propagated), and `x, y` are incremented, then they are assigned to `y, z`, respectively. Observe that the value of `x` read at any positive edge of the `clock` is incremented and then written to `y`. This new value of `y` is going to be read at the next positive edge of the `clock`, which implements a registered incrementer. This is repeated for the value of `y` and the resultant value in `z`.
- The order in which the three statements are specified in the Verilog specification is irrelevant.

Notice that the input is changed at time `10`; however, neither of the `x,y,z` values change at time `10`. This is the correct behaviour because `x,y,z` are all registers. Therefore, we expect `x` to show the value `1` at time `20`, which is the case. At time `30` the value registered in `x` is read by the first incrementer, and written to `y`. This continues for `z`. Notice that it takes three clock cycles for the value of `3` to be available in `z`. This is the expected behaviour.

non-blocking.v

```
1  // Example:
2  // in->|x| -> +1 -> |y| -> +1 -> |z| ->out
3  module non_blocking
4    (
5     input  clock,
6     input [3:0]  in,
7     output [3:0] out
8     );
9     // Internal registers
10    reg [3:0]   x,y,z;
11    always @(posedge clock) begin
12       x <= in;
13       y <= x + 1;
14       z <= y + 1;
15       $display("[blk] time:%t, in: %h, x: %h, y:%h, z:%h",$time, in, x,y,z);
16    end
17    assign out = z;
18
19    always @(x) begin
20       $display("time: %t, -- x:%h, clock: %h", $time, x, clock);
21    end
22 endmodule // non_blocking
23
24 module dut;
25    reg [3:0] in;
26    reg clock;
27    wire [3:0] out;
28
29    always begin
30       #5 clock = ~clock;
31    end
32    non_blocking nb( .clock(clock), .in(in),.out(out) );
33    initial begin
34   $dumpfile("non-blocking.vcd");
35   $dumpvars(0,nb);
36       #0 clock = 1;
37       #0 in = 3'h1;
38       #100 $finish;
39    end
40 endmodule // dut
```

```
[blk] time:                   0, in: x, x: x, y:x, z:x
[blk] time:                  10, in: 1, x: x, y:x, z:x
[blk] time:                  20, in: 1, x: 1, y:x, z:x
[blk] time:                  30, in: 1, x: 1, y:2, z:x
[blk] time:                  40, in: 1, x: 1, y:2, z:3
[blk] time:                  50, in: 1, x: 1, y:2, z:3
[blk] time:                  60, in: 1, x: 1, y:2, z:3
[blk] time:                  70, in: 1, x: 1, y:2, z:3
[blk] time:                  80, in: 1, x: 1, y:2, z:3
[blk] time:                  90, in: 1, x: 1, y:2, z:3
[blk] time:                 100, in: 1, x: 1, y:2, z:3
[blk] time:                 110, in: 1, x: 1, y:2, z:3
```

Listing 14: Example of a pipeline using `non-blocking` assignments.

Exercise 3.1 Order of non-blocking statements

Please explain why the order in which the three non-blocking statements specified within the `always` is irrelevant? Use no more than 5 sentences.

Now suppose that we change the **non-blocking** assignments to **blocking** assignments, and those are the only changes we make to this example. This amounts to changing the <= to =. The output of the simulation is shown in Listing 15.

blocking.v

```
1  // Example: in->|x| -> +1 -> |y| -> +1 -> |z| ->out
2  module blocking
3    (
4     input  clock,
5     input [3:0]  in,
6     output [3:0] out
7     );
8
9     reg [3:0]   x,y,z;
10    always @(posedge clock) begin
11       x = in;
12       y = x + 1;
13       z = y + 1;
14       $display("[blk] time:%t, in: %h, x: %h, y:%h, z:%h",$time, in, x,y,z);
15    end
16    assign out = z;
17 endmodule // blocking
```

```
[blk] time:                   0, in: x, x: x, y:x, z:x
[blk] time:                  10, in: 1, x: 1, y:2, z:3
[blk] time:                  20, in: 1, x: 1, y:2, z:3
[blk] time:                  30, in: 1, x: 1, y:2, z:3
[blk] time:                  40, in: 1, x: 1, y:2, z:3
[blk] time:                  50, in: 1, x: 1, y:2, z:3
[blk] time:                  60, in: 1, x: 1, y:2, z:3
[blk] time:                  70, in: 1, x: 1, y:2, z:3
[blk] time:                  80, in: 1, x: 1, y:2, z:3
[blk] time:                  90, in: 1, x: 1, y:2, z:3
[blk] time:                 100, in: 1, x: 1, y:2, z:3
[blk] time:                 110, in: 1, x: 1, y:2, z:3
```

Listing 15: Example of **WRONG** pipeline implementation using `blocking` assignments.

Exercise 3.2 Very bad sequential hardware

Draw a hardware diagram (using similar blocks as Figure 3) to describe the resultant hardware for the **blocking** version of the pipeline specification as shown in Listing 15. Using no more than 5 sentences, please explain what the resultant hardware does.

3.4.2 Register with AND gate

Now that we understand how to build pipelines, we can extend our AND gate example from earlier and register its output. Note that this is an example of how each stage in the pipeline would behave: some computation is done using combinational circuits (AND gate in this example), and stored in a sequential component (a register in this example).

- Whenever there are sequential elements, we must be sure to reset the state element. You learned in ECE 327 how to write specifications for reset. This example shows how to implement synchronous resets. Resets are essential to ensure that the state elements are in a known state prior to starting the operation of the hardware.

Exercise 3.3 Synchronous versus asynchronous reset

Write the Verilog specification of the example in Listing 16 to implement the registered AND gate with an asynchronous reset. Using no more than 5 sentences, describe the difference between asynchronous and synchronous resets.

Exercise 3.4 Registered input to AND gate

Write the Verilog specification using the example in Listing 16 to register both the inputs and outputs to the AND gate. Make sure to also show the output of the simulation. Explain whether the simulation output is the expected behaviour or not.

reg-and.v

```
module and_assign ( x, y, z );
   input wire x, y;
   output wire z;
   assign z = x & y;
endmodule // and_assign

module reg_and
  (
   input wire  clock,
   input wire  reset,
   input wire  in,
   output   out
   );

   // Sequential logic
   reg     out;

   always @(posedge clock )  begin
      if ( reset )
        out <= 0;
      else
        out <= in;
   end

//   assign out = out;
endmodule // reg_and

module dut;
   reg x, y, reset;
   reg clock = 1;
   wire z, out;

   initial begin
      $dumpfile("reg-and.vcd");
      // All variables (0) from module instance gate
      $dumpvars(0, gate);
      $dumpvars(0, reg_gate);

      #0 reset = 1;
      #20 x = 0; y = 0;
       reset <= 0; $display("Reset complete");
      #10 x = 1; y = 1; $display("set 1 1");
      #10 x = 1; y = 0;
      #10 x = 1; y = 1;
      #10 x = 0; y = 1;
      #20 $finish;
   end
   // Instantiate AND gate
   and_assign gate ( .x(x), .y(y), .z(z));
   reg_and    reg_gate ( .clock(clock), .reset(reset), .in(z), .out(out) );
   // Toggle clock signal every 5
   always begin
      #5 clock = ~clock;
   end
   always @(posedge clock) begin
      $display("time=%t, reset=%b, x=%b, y=%b, z=%b, out=%b", $time,reset,x,y,z, out);
   end
endmodule // dut
```

Listing 16: Register with AND Gate: Combinational using **always**.

4 Summary

You made it to the end of the tutorial. :)