

Verilog and Basic Syntax of Verilog

Verilog is a hardware description language (HDL) which has C-like concise syntax. It has simple module/port syntax so that we can organize hierarchical building blocks and manage complexity.

In this tutorial, all codes can be found in repository:

<https://github.com/2110363-HW-SYN-LAB/verilog-syntax-examples/tree/main>

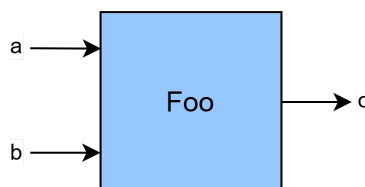
Basic Building Block: Module

The module is the basic building block in Verilog. It can be interconnected to describe the structure of the digital system. Modules start with a keyword `module` and end with a keyword `endmodule`.

```
module <module_name> (<port_list>);  
  
    ...  
  
endmodule
```

The module consists of module name, port list, and can contain other module instances inside the module. For the ports, ports are similar to pins on a chip. It provides a way to communicate with the outside world. Ports can be input, output, and inout.

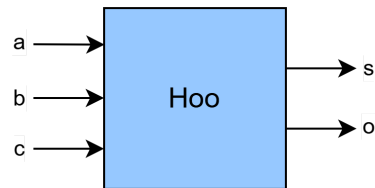
For example, the module `Foo` has 2 input ports (`a` and `b`) and 1 output port (`o`). Inside the module, it performs some logic on 2 input ports (which will be discussed later). We can declare modules in 2 ways, declaring port type outside the port list, or declaring port type inside the port list.



```
module Foo(a, b, o);  
    input a, b;  
    output o;  
  
    ...<some logic here>...  
  
endmodule
```

```
module Foo(  
    input a,  
    input b,  
    output o  
);  
  
    ...<some logic here>...  
  
endmodule
```

For another example, the module Hoo has 3 input ports (a, b, and c) and 2 output ports (s and o). Inside this module, there is some logic which will perform something.



```
module Hoo(a, b, c, s, o);
    input a, b, c;
    output s, o;

    ...<some logic here>...

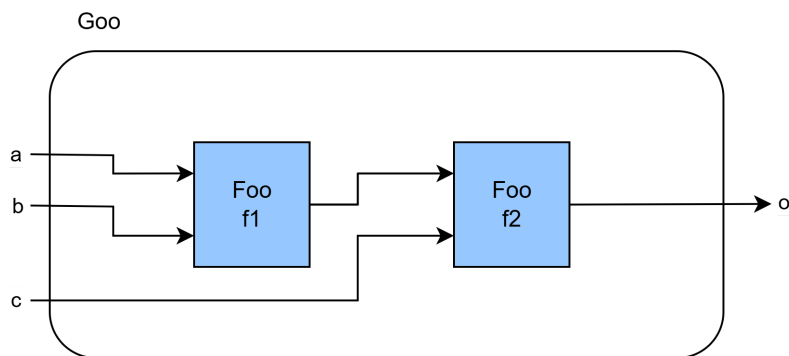
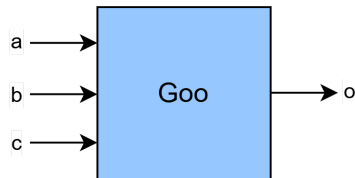
endmodule
```

```
module Hoo( input a, b, c
            output s, o );

    ...<some logic here>...

endmodule
```

Also, the modules can contain other module instances inside. For example, the module Goo contains 2 instances of module Foo as shown in the figure below.



```
module Goo (a, b, c, o);
    input a, b, c;
    output o;

    wire temp;

    Foo f1 (
        .a(a),
        .b(b),
        .o(temp)
    );
    Foo f2 (
        .a(temp),
        .b(c),
        .o(o)
    );

endmodule
```

Data Types

There are 2 main data types in Verilog:

1. Nets

Net data types represent physical connections, like wires, in a circuit. They do not store values but instead pass values from one place to another. They always reflect the logic value of the driving devices. There are many types of nets, but we mostly use **wire**. To declare a wire, use the following syntax:

```
wire [<range>] <net_name>;
```

We can specify a range as [MSB:LSB]. The default is 1 bit wide.

For example, the following codes show how to declare wires:

```
wire w1;           // 1-bit wire named w1
wire [3:0] w2;      // 4-bit wire named w2
```

2. Variables

A variable is an abstraction of a data storage and can hold values. The most popular variable used is **reg**, which is used for storing values in sequential logic (flip-flops, latches) or in combinational logic when a procedural block assigns values. Note that reg does not necessarily imply a hardware register. To declare a reg, use the following syntax:

```
reg [<range>] <reg_name>;
```

For example, the following codes show how to declare regs:

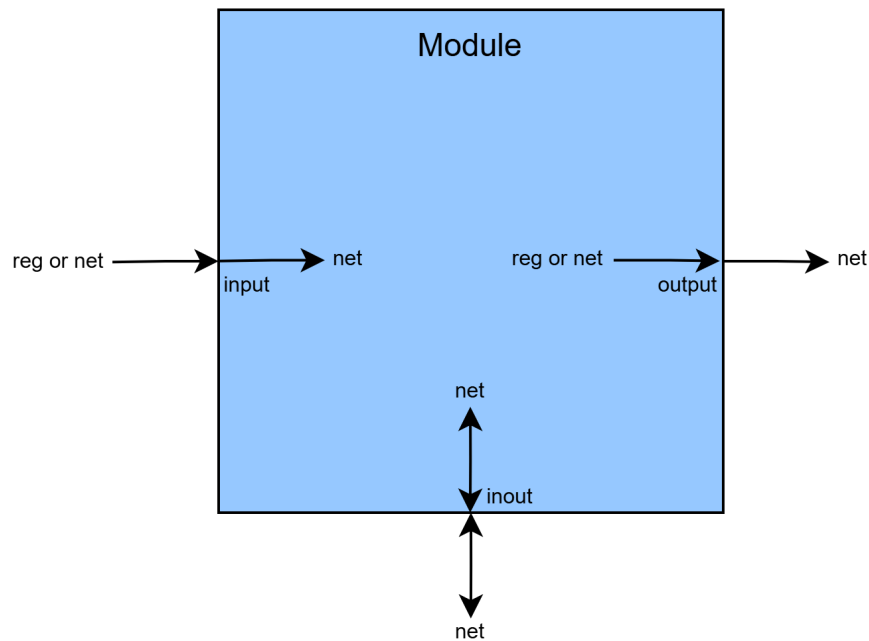
```
reg r1;           // 1-bit reg named r1
reg [7:0] r2;      // 8-bit reg named r2
```

Also, reg can be used to declared a memory using the following syntax:

```
reg [<range>] <memory_name> [<start_addr>:<end_addr>];

// Example
reg [7:0] mem [0:1023]    // a 1-KB memory
```

We can visualize a port as consisting of two units, one unit that is internal to the module another that is external to the module. The internal and external units are connected. There are rules governing port connections when modules are instantiated within other modules. The Verilog simulator complains if any port connection rules are violated. These rules are summarized in the figure below.



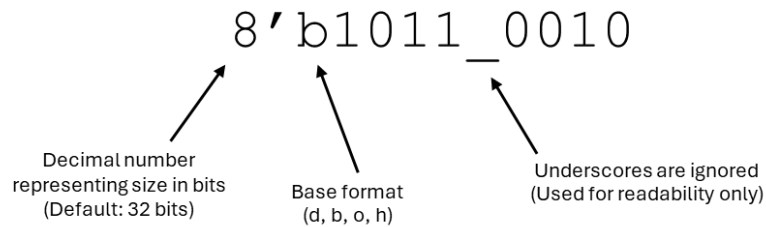
Data Values

There are 4 types of data values in Verilog:

| Data Values | Descriptions |
|-------------|--|
| 0 | zero, logic low, false, ground |
| 1 | one, logic high, power |
| x | unknown |
| z | high impedance, unconnected, tri-state |

Numeric Literals

In Verilog, numbers can be assigned with size or not. If it is not declared the size it defaults to 32 bits. Also you must choose the base of the number and the options are decimal, hexadecimal, octal and binary. If not specified the base it defaults to the decimal base.



Examples:

```
123          // 32-bit wide with the value of 123 in decimal base
4'b1011      // 4-bit wide with the value of 1011 in binary base
4'b1_011     // same as 4'b1011
8'h4A        // 8-bit wide with the value of 4A in hexadecimal base
6'o77        // 6-bit wide with the value of 77 in octal base
```

Operators

| Operators | Descriptions |
|----------------------|---------------------------|
| Arithmetic Operators | |
| a + b | a plus b |
| a - b | a minus b |
| a * b | a multiplied by b |
| a / b | a divided by b |
| a % b | a modulo b |
| a ** b | a to the power of b |
| Relational Operators | |
| a < b | a less than b |
| a > b | a more than b |
| a <= b | a less than or equal to b |
| a >= b | a more than or equal to b |
| Equality Operators | |

| Operators | Descriptions |
|---|--------------------------------------|
| <code>a === b</code> | a equal to b (including x and z) |
| <code>a !== b</code> | a not equal to b (including x and z) |
| <code>a == b</code> | a equal to b |
| <code>a != b</code> | a not equal to b |
| Logical Operators | |
| <code>a && b</code> | return true if a and b are true |
| <code>a b</code> | return true if a or b are true |
| <code>!a</code> | not a |
| Bitwise Operators | |
| <code>a & b</code> | a AND b |
| <code>a b</code> | a OR b |
| <code>a ^ b</code> | a XOR b |
| <code>~a</code> | NOT a |
| <code>a ~^ b</code> | a XNOR b |
| Reduction Operators (perform bitwise on all bits) | |
| <code>&a</code> | Perform AND on all bits on a |
| <code> a</code> | Perform OR on all bits on a |
| <code>~&a</code> | Perform NAND on all bits on a |
| <code>~ a</code> | Perform NOR on all bits on a |
| <code>^a</code> | Perform XOR on all bits on a |
| <code>~^a</code> | Perform XNOR on all bits on a |
| Shift Operators | |
| <code><<</code> | Logical left shift |
| <code>>></code> | Logical right shift |
| <code><<<</code> | Arithmetic left shift |
| <code>>>></code> | Arithmetic right shift |
| Concatenation and Replication Operators | |
| <code>{a,b}</code> | Concatenate a and b |
| <code>{b{a}}</code> | Replicate a for b times |

Continuous Assignment

Continuous assignment is a construct used to drive values onto nets (wires) using a combinational logic expression. It is typically used to model hardware components where outputs are continuously determined by the inputs, without any storage or clocking.

The `assign` keyword is used for continuous assignments. The general syntax is:

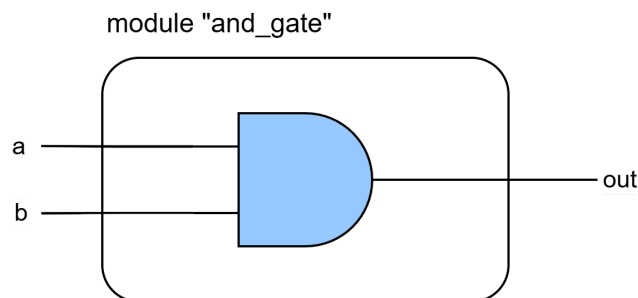
```
assign <net_name> = <expression>;
```

Example 1: AND Gate

Code can be found in:

https://github.com/2110363-HW-SYN-LAB/verilog-syntax-examples/tree/main/ex01_and_gate

Let's start with a simple design of a digital circuit. We want to perform AND operation on 2 input ports. Therefore, we need 2 input ports (let them be `a` and `b`) and 1 output port for the result from performing AND operation. The design is as shown below:



Now, we will use continuous assignment to assign the output wire `out` so that `out=a&b`. Therefore, the code in `and_gate.v` will be written as follow:

```
module and_gate(a, b, out);  
    input a, b;  
    output out;  
  
    assign out = a & b;  
  
endmodule
```

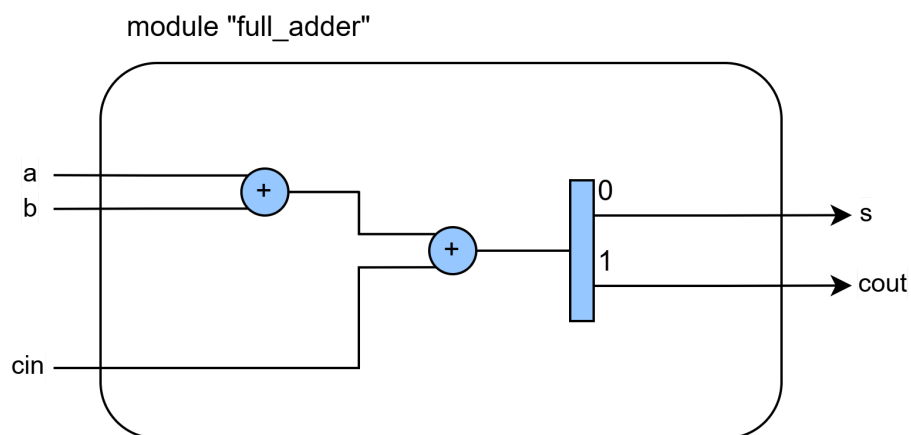
You can run `and_gate_tb.v` for testbench (using Xilinx simulator or compiling with Icarus Verilog and simulating with GTKWave).

Example 2: 1-bit Full Adder

Code can be found in:

https://github.com/2110363-HW-SYN-LAB/verilog-syntax-examples/tree/main/ex02_1_bit_full_adder

Now, let's design something more interesting like an adder circuit. We know that the maximum size of result from adding three of 1-bit input is 2 bits. Therefore, our design must have 3 input ports (a, b, and cin) and 2 output ports (s and cout). We can design the circuit using logic gates, but in Verilog, there are arithmetic operations to use so that our life is easier. Also, we can use concatenation operations to merge two 1-bit wires into 2-bit wire so that a carry from addition will be at MSB of 2-binary-digit results.



Therefore, our code in full_adder.v must have concatenation of s and cout {cout, s} and assign the addition result to this.

```
module full_adder (a, b, cin, s, cout);  
    input a;  
    input b;  
    input cin;  
    output s;  
    output cout;  
  
    assign {cout, s} = a + b + cin;  
  
endmodule
```

For example, if $a = 0$, $b = 1$, and $cin = 1$, then $\{cout, s\} = 0 + 1 + 1 = 10$ in binary system. So, cout will be 1 and s will be 0.

You can run full_adder_tb.v for testbench (using Xilinx simulator or running with Icarus Verilog and simulating with GTKWave).

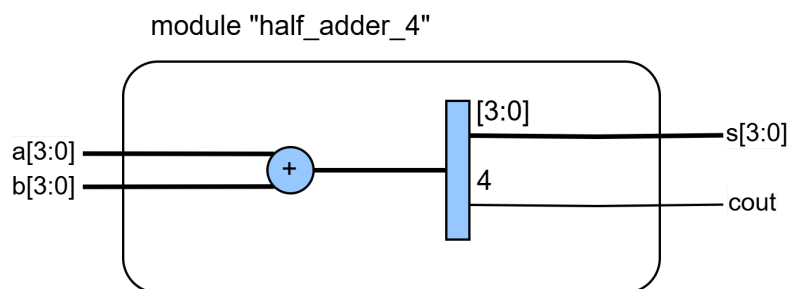
Example 3: 4-bit Half Adder

Code can be found in:

https://github.com/2110363-HW-SYN-LAB/verilog-syntax-examples/tree/main/ex03_4_bit_half_adder

When our inputs or outputs are larger than 1 bit, we can specify how wide the port be like what we do when declaring wire or reg with specific range (which is adding range in front of port name).

Our design is similar to example 2, but we need to specify the ranges of ports. Our result from addition will be 5 binary digits which will be assigned to `cout` for MSB and `s` for the other 4 LSBs.



Therefore, our code in `half_adder_4.v` must have concatenation of `s` and `cout` `{cout, s}`, and assign the addition result to this.

```
module half_adder_4(a, b, s, cout);  
    input [3:0] a;  
    input [3:0] b;  
    output [3:0] s;  
    output cout;  
  
    assign {cout, s} = a + b;  
  
endmodule
```

You can run `half_adder_4_tb.v` for testbench (using Xilinx simulator or running with Icarus Verilog and simulating with GTKWave).

Initial and Always Processes

Initial and always statements describe independent processes, meaning that the statements in one process execute autonomously. Both types of processes consist of procedural statements (which will be discussed in the next topic) and both start immediately as the simulator is started. In the module, it is possible to have multiple initial and always statements.

- For initial statements, the process will be executed only once when simulation is started ($t=0$).

```
initial
begin
    ...

end
```

For example, if we want to set the initial value of reg a to 3, then assigning value 3 into a will be in an initial block.

```
initial
begin
    a = 3;
end

// OR

initial a = 3;
```

- For always statements, the process will be always executed when the simulation is started ($t=0$) and repeatedly executed when the signal in the sensitivity list is changed.

```
always @(<sensitivity_list>)
begin
    ...

end
```

For example, the following always block will be executed when a or b is changed.

```
always @(a, b) begin
    c = a & b;
end

// OR

always @(a or b) begin
    c = a & b;
end
```

In Verilog-2001, we can use the @(*) construct which creates a sensitivity list for all signals read in the always block. For example, the following always block will be executed when a or b or c is changed.

```
always @(*)
begin
    s = a & b & c;
end
```

Procedural Assignment

Procedural assignment is used within procedural blocks, such as always or initial, to assign values to variables like reg. Unlike continuous assignment (which is used for nets), procedural assignments describe behavior that changes conditionally and is often used to model sequential or combinational logic.

There are 2 types of procedural assignments:

1. Blocking Assignment (=)

For this type of assignment, the statements will execute sequentially within a block. The statement must complete before the next statement is executed. It is commonly used in combinational logic.

Example:

```
always @(posedge clk) begin
    a = b;          // Blocking assignment
    c = a + 1;      // Depends on the new value of a
end
```

2. Non-blocking Assignment (<=)

For this type of assignment, it allows assignments to occur in parallel. The expression on the right-hand side is evaluated, but the assignment happens at the end of the block. It can be used in sequential logic to model flip-flop behavior. Also, it can avoid race conditions when modeling synchronous systems.

Example:

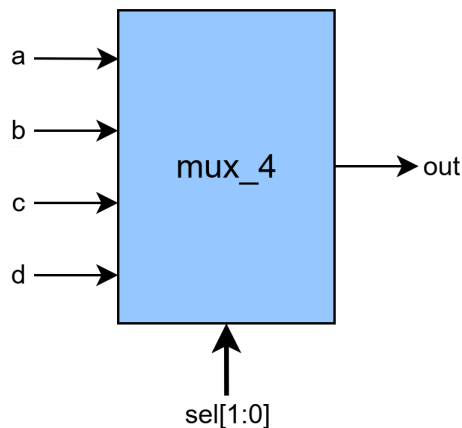
```
always @(posedge clk) begin
    a <= b;          // Non-blocking assignment
    c <= a + 1;      // Depends on the old value of a
end
```

Example 4: 4-input Multiplexer

Code can be found in:

https://github.com/2110363-HW-SYN-LAB/verilog-syntax-examples/tree/main/ex04_4_input_mux_4_iplexer

Let's design a multiplexer using always block and procedural assignment. Our multiplexer requires 4 inputs and a 2-bit selector to select the input. We can design the multiplexer using combinatorial logic, but in Verilog, we can use if-else statements and switch-case statements. From the design below, our module has 5 input ports (a, b, c, d, and sel) and 1 output port (out).



We will use a case statement to choose an input depending on the selector. The case statement is used for conditional branching based on the value of an expression. It works like a switch-case statement in other programming languages and is particularly useful for implementing multiplexers, decoders, and other combinational logic. In mux_4.v, the code includes a case statement depending on sel. By the way, what happens when sel==xx? That's why there is a default case so that out will be unassigned. Note that LHS of procedural assignment must be reg.

```
module mux_4 (
    input a, b, c, d,
    input [1:0] sel,
    output reg out
);

    always @(*) begin
        case(sel)
            2'b00: out = a;
            2'b01: out = b;
            2'b10: out = c;
            2'b11: out = d;
            default: out = 1'bx;
        endcase
    end
endmodule
```

You can run mux_4_tb.v for testbench (using Xilinx simulator or running with Icarus Verilog and simulating with GTKWave).

You can implement multiplexer in another way by using an if-else statement, which is similar to the case statement above. This code is in mux_4_ifelse.v.

```
module mux_4_ifelse (
    input a, b, c, d,
    input [1:0] sel,
    output reg out
);

    always @(*) begin
        if (sel == 2'b00) begin
            out = a;
        end else if (sel == 2'b01) begin
            out = b;
        end else if (sel == 2'b10) begin
            out = c;
        end else if (sel == 2'b11) begin
            out = d;
        end else begin
            out = 1'bx;
        end
    end
end

endmodule
```

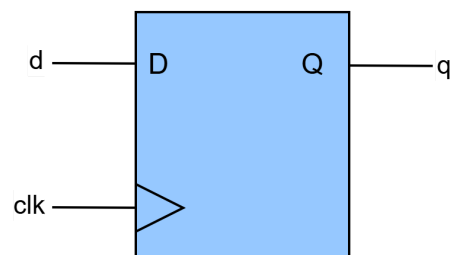
You can run mux_4_ifelse_tb.v for testbench (using Xilinx simulator or running with Icarus Verilog and simulating with GTKWave).

Example 5: D Flip Flop

Code can be found in:

https://github.com/2110363-HW-SYN-LAB/verilog-syntax-examples/tree/main/ex05_dff

D flip-flop is a sequential logic circuit that captures and stores the value of a data input at the rising or falling edge of a clock signal. The stored value remains constant until the next clock edge. In this example, we will focus on basic rising-edge D flip flop without reset signal.



Our module must have 2 input ports, which are `d` for data input and `clk` for clock signal. Also, we have only 1 output port (`q` for stored data). We want to assign the `q` to be as same as `d` every rising edge of the clock. So our procedural block (`always` block) must be executed at every rising edge of the clock. We can represent the rising edge of the signal by using keyword “`posedge <signal>`” and represent the falling edge of the signal by using keyword “`negedge <signal>`”. Therefore, our sensitivity list will be `posedge clk` (triggered every rising edge of clock signal).

```
module dff (  
    output reg q,  
    input wire d,  
    input wire clk  
);  
  
    always @(posedge clk) begin  
        q <= d;  
    end  
  
endmodule
```

Parameterized Modules

Parameterized modules allow you to write flexible and reusable code by defining parameters that can be customized during module instantiation. This is particularly useful for designing hardware components like memories, counters, or multiplexers with configurable widths, depths, or other properties. The syntax for declaring parameters is:

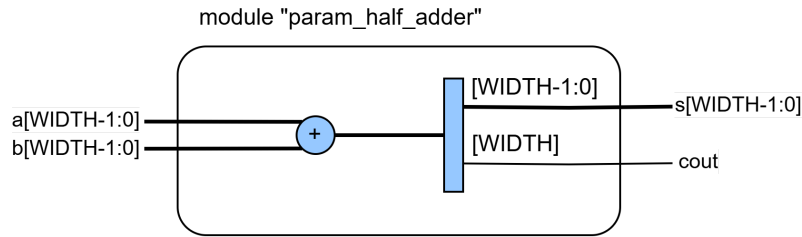
```
module <module_name> #(<parameters>) (<port_list>);  
  
    ...  
  
endmodule
```

Example 6: Parameterized Half Adder

Code can be found in:

https://github.com/2110363-HW-SYN-LAB/verilog-syntax-examples/tree/main/ex06_parameterized_half_adder

We have been designed-bit half adder in example 3. By the way, if we want to design N-bit half adders, we may need to create each module (like 1-bit, 2-bit, 3-bit, ...) which isn't practical. We will design our half adder that can work with any range of input. Let's the width of the input (the number of bits of input be `WIDTH`), then our design will be like below:



We call this `WIDTH` as a parameter. Note that inputs' ranges must be specified as `[WIDTH-1:0]` so that the number of bits of any input is `WIDTH`. The code is similar to example 3, but we will change the MSB of all ranges to `WIDTH-1`.

```
module param_half_adder #(
    parameter WIDTH = 1
) (
    input wire [WIDTH-1:0] a,
    input wire [WIDTH-1:0] b,
    output wire [WIDTH-1:0] s,
    output wire cout
);

    assign {cout, s} = a + b;

endmodule
```

In the parameter list, `WIDTH = 1` means the default value of `WIDTH` is 1.

If we want to initialize this module instance, just do like what we did before, but we must add the parameter in front of the instance name. For example:

```
// Initializing instance named ha with WIDTH=2
param_half_adder #(.WIDTH(2)) ha (a, b, s, cout);

// Initializing instance named ha2 with WIDTH=4
param_half_adder #(.WIDTH(4)) ha2 (a, b, s, cout);

// Initializing instance named ha3 with WIDTH=1 (default value)
param_half_adder ha3 (a, b, s, cout);
```

Simulating Designs

After designing and implementing modules, we need to check if our implementations work well. This can be done by simulating our designs. First, we need to create our simulations in Verilog files to check our designs, we will call them “testbenches”. These files will assign the values of inputs of the module and show the outputs in waveforms. Let’s see example 1 in the previous tutorial.

Example code can be found in:

https://github.com/2110363-HW-SYN-LAB/verilog-syntax-examples/tree/main/ex01_and_gate

We will create a Verilog file for testbench, let’s name it `and_gate_tb.v`.

```
`timescale 1ns/1ns  Specifies the time unit and time precision for the simulation

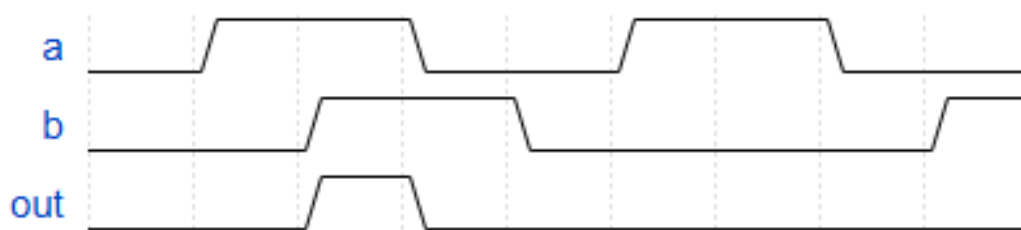
module and_gate_tb();
    reg a, b;
    wire out;

    and_gate uut(a, b, out);  Our instance of module we want to test

    initial begin
        $dumpfile("and_gate_tb.vcd");  Waveform Dump
        $dumpvars(0, and_gate_tb);

        a = 0;
        b = 0;
        #10 a = 1;  #10: Delay of 10 seconds to show in waveforms
        #10 b = 1;
        #10 a = 0;
        #10 b = 0;
        #10 a = 1;
        #10 b = 0;
        #10 a = 0;
        #10 b = 1;
        #10 $finish;  Ending the simulation
    end
endmodule
```

The above testbench is equivalent to the following waveform:

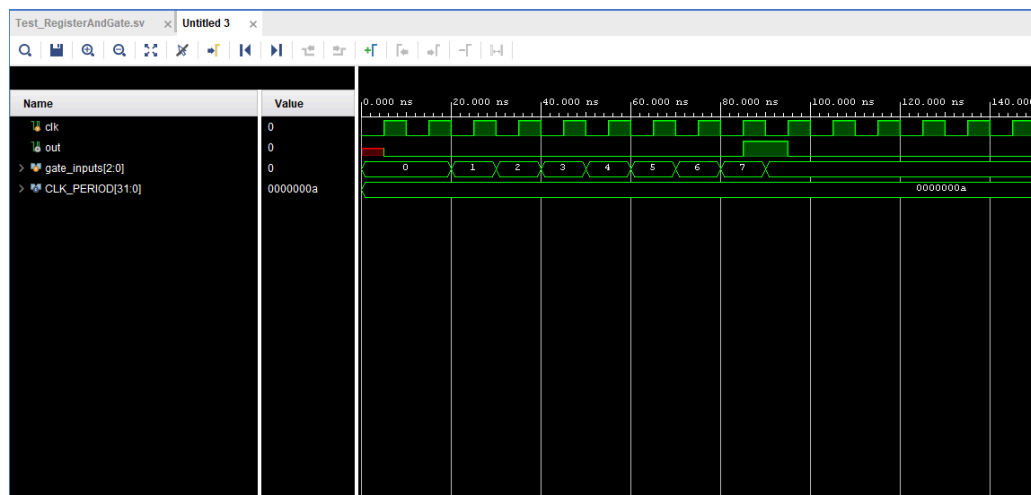
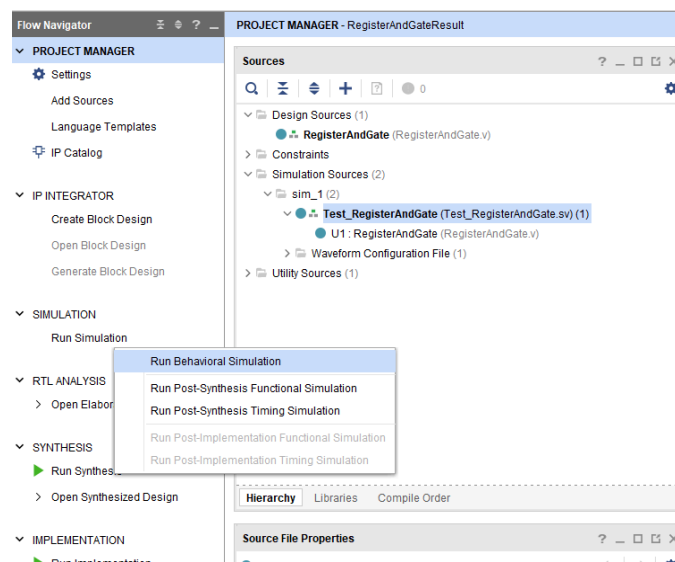


Simulating using Vivado Simulator

Vivado has a built-in simulator which supports functional and timing simulations for VHDL, Verilog, SystemVerilog (SV), and mixed VHDL/Verilog or VHDL/SV designs. According to [Vivado Design Suite User Guide for Simulation](#), the Vivado simulator supports the following features:

- Source code debugging (step, breakpoint, current value display)
- SDF annotation for timing simulation
- VCD dumping
- One-click compilation and simulation
- Built-in support for Xilinx simulation libraries
- Real-time waveform update
- etc.

To run the simulation, we must select our testbench to be a top-level module. From the Flow Navigator, click **Run Simulation** and select a simulation type (in this case, we will select **Run Behavioral Simulation**) to invoke the Vivado simulator workspace.



Simulating using Icarus Verilog and GTKWave

Icarus Verilog is an open-source Verilog simulation and synthesis tool. It is widely used for simulating Verilog designs, particularly in academic, open-source, and lightweight development environments.

GTKWave is an open-source waveform viewer used to analyze and debug signal transitions in hardware designs. It is often paired with simulators like Icarus Verilog to visualize the behavior of Verilog and VHDL designs.

To simulate using Icarus Verilog and GTKWave, firstly, you must install these tools:

For Windows

You can download Icarus Verilog with GTKWave from this [website](#).

For MacOS

You can install Icarus Verilog using Homebrew:

```
brew install icarus-verilog
```

Sadly, some MacOS do not support GTKWave. Try these methods to install GTKWave:

- Try to install GTKWave by following these steps in the [official wiki](#). You may have to compile GTKWave by yourself.
- If it does not work, you may use other applications such as [Surfer](#). You can import .vcd files (which will get from compiling the Verilog codes from Icarus Verilog).

For Linux

You can install Icarus Verilog and GTKWave using package manager.

```
sudo apt install iverilog gtkwave
```

The flow of compiling and visualizing our designs will be divided into 3 parts:

1. Compiling the code

We must compile our designs first by using Icarus Verilog:

```
iverilog -o output_file teshbench.v design.v
```

Where `output_file` is the name of a compiled file (in vvp format), and `testbench.v` and `design.v` are our Verilog codes.

2. Running the simulation

We must execute the compiled file using `vvp` command:

```
vvp output_file
```

This will run our testbench and dump the waveforms generated in the testbench (\$dumpfile and \$dumpvars) in vcd format.

3. Analyzing waveforms

After the waveforms are dumped, we can watch our generated waveforms using GTKWave:

```
gtkwave output.vcd
```

Running this will show the waveforms of `output.vcd` in the GTKWave program.

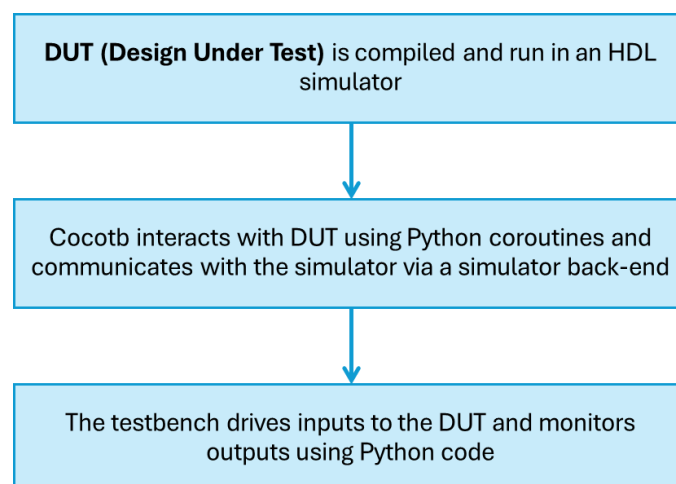
Verifying and Testing with Cocotb

Verifying and testing are critical in Verilog (and hardware design in general) because hardware designs, once implemented in silicon, cannot be easily modified. Mistakes in design can lead to costly errors, defective products, or catastrophic failures. Verification ensures the design behaves as intended under all scenarios, making it a key part of the development lifecycle.

Introduction to Cocotb

[Cocotb](#) (Coroutine-based Co-simulation Testbench) is a Python-based testing framework for verifying hardware designs written in HDL (such as Verilog or VHDL). It allows users to write testbenches in Python, leveraging its simplicity, extensive libraries, and ecosystem to create powerful and flexible verification environments.

The workflow of Cocotb is as the following:



Cocotb Installation

Because Cocotb is a Python library that can be worked in Python 3.6+, there are many ways to install. By the way, the requirements are:

- Python 3.6+
- GNU Make 3+
- A Verilog or VHDL simulator/compiler (e.g. Icarus Verilog)

For more information, you can visit this [website](#).

Installing Cocotb using Conda

Conda is an open-source package management system and environment management tool that allows users to install, run, and update software and manage multiple development environments efficiently. It is popular among data scientists, developers, and researchers for handling libraries, dependencies, and project isolation.

1. Install Miniconda by following steps on this [official website](#) (Some of you may have already installed Miniconda, you can skip this step). To verify your installation, you can check by running this command in your terminal, it should show your installed Conda's version.

```
conda --version
```


2. Create a new environment, let's name it "cocotb", with Python version 3.10.

```
conda create -n cocotb python=3.10
```

3. Activate your created environment.

```
conda activate cocotb
```

Now, your prompt should have an environment name at the front like this.



```
(cocotb) C:\Users\soraw>
```

4. Install tools related to GNU Make.

```
conda install -c msys2 m2-base m2-make
```

5. Install Cocotb and Pytest libraries.

```
pip install cocotb pytest
```

Running Cocotb

To run testing with Cocotb, we need 3 files:

- **Your HDL designs** (e.g. Verilog files)
- **Cocotb testbenches in Python files** which contain test cases and coroutines to drive the simulation and check DUT behavior.
- **Makefile** for managing the simulation and connecting the DUT with the Cocotb testbench.

Let's try running the Cocotb testbench in example 1.

Example code can be found in:

https://github.com/2110363-HW-SYN-LAB/verilog-syntax-examples/tree/main/ex01_and_gate

If you clone the repository, you will notice that there is a test directory inside each example, that's where our Cocotb testbenches will be in. First, we need to create a Cocotb testbench for verifying and testing our design. Let's name it `and_gate_test.py`.

```
import cocotb
from cocotb.triggers import Timer

@cocotb.test()
async def and_gate_test(dut):
    a = (0, 0, 1, 1)
    b = (0, 1, 0, 1)
    out = (0, 0, 0, 1)

    for i in range(4):
        dut.a.value = a[i]
        dut.b.value = b[i]
        await Timer(1, units='ns')
        assert dut.out.value == out[i], f"[Iteration {i}] output is {dut.out.value}, expected {out[i]}"
```

We will assign the value of inputs by using `dut.<input_name>.value = <value>` and retrieve that value of output from `dut.<output_name>.value`. Like other Python applications, we can test our design by using `assert` keyword. For creating a test case, you must create a function and add decorator `@cocotb.test()` above the function. Note that you can create as many test cases as you want. To simulate the timer, we will use `Timer` which will fire after the specified simulation time period has elapsed.

If your design involves clock, you can use `Clock`, `RisingEdge`, and `FallingEdge` classes to simulate your clock, which can be found in example 5 in the same GitHub repository.

Next, we will create `Makefile`, which will perform simulation managing automation. (The code below show only an important part of `Makefile`)

```

# Simulator
SIM ?= icarus

# Top level language
TOPLEVEL_LANG ?= verilog

# Files
VERILOG_SOURCES = $(shell pwd)/<source_path>

# TOPLEVEL is the name of the toplevel module in your Verilog file
TOPLEVEL = <toplevel_name>

# MODULE is the basename of the Python test file
MODULE = <cocotb_python_file_name>

# include cocotb's make rules to take care of the simulator setup
include $(shell cocotb-config --makefiles)/Makefile.sim

```

You must specify each environment variables:

- **SIM**: The simulator you will use. In this tutorial, we will use Icarus Verilog.
- **TOPLEVEL_LANG**: The language of your design, which is Verilog.
- **VERILOG_SOURCES**: Your file paths of Verilog designs.
- **TOPLEVEL**: Your top level module name of your design inside Verilog source files.
- **MODULE**: Your Cocotb Python file path.

Let's run our testbench by running the following command in your terminal in test directory. Don't forget to activate your Conda environment if you use Conda or Miniconda.

```
make
```

This command will run your Makefile and run your Cocotb testbench. If it goes well, it should show the statistics and details of your test.

```

(cocotb) C:\dev\verilog\ex01_and_gate\test>make
rm -f results.xml
"make" -f Makefile results.xml
make[1]: Entering directory '/c/dev/verilog/ex01_and_gate/test'
rm -f results.xml
MODULE=and_gate_test TESTCASE= TOPLEVEL=and_gate TOPLEVEL_LANG=verilog \
/c/verilog/bin/vvp -R C:/Users/soraw/miniconda3/envs/cocotb/Lib/site-packages/cocotb/libs -m cocotbvpi_icarus sim_build/sim.vvp
-.-ns INFO gpi ..mbed\gpi_embed.cpp:81 in set_program_name_in_venv Did not detect Python virtual environment. Using syst
sm-wide Python interpreter
-.-ns INFO gpi ..\gpi\GpiCommon.cpp:101 in gpi_print_registered_impl VPI registered
0.00ns INFO cocotb Running on Icarus Verilog version 12.0 (devel)
0.00ns INFO cocotb Running tests with cocotb v1.9.2 from C:\Users\soraw\miniconda3\envs\cocotb\lib\site-packages\cocotb
0.00ns INFO cocotb Seeding Python random module with 1734686025
0.00ns INFO cocotb.regression Found test and_gate_test and_gate_test
4.00ns INFO cocotb.regression *{34mrunning*{49m*{39m and_gate_test (1/1)
4.00ns INFO cocotb.regression and_gate_test *{32mpassed*{49m*{39m
*****
** TEST STATUS SIM TIME (ns) REAL TIME (s) RATIO (ns/s) **
*****
** and_gate_test.and_gate_test *{32m PASS *{49m*{39m 4.00 0.00 inf **
*****
** TESTS=1 PASS=1 FAIL=0 SKIP=0 4.00 0.04 101.15 **
*****
make[1]: Leaving directory '/c/dev/verilog/ex01_and_gate/test'

```

Constraints Files

To program your board, you need to inform the software what physical pins on the board that you plan on using or connecting to in relation to the code that you design to describe the behavior of the FPGA.

XDC Files

The constraints files that Xilinx's Vivado uses are called XDC files (Xilinx Design Constraints file). They are used in Xilinx FPGA design workflows to specify design constraints. These constraints guide the synthesis, placement, and routing processes, ensuring the design meets functional, timing, and physical requirements. For Basys 3 FPGA board, the example of the constraints file is in the master branch of [official GitHub repository](#).

In the XDC file, we must define the constraints we want to use in our design. For each constraint, we must define the name of external FPGA pin, the electrical standard used, and the name which match the input or output ports in the Verilog code.

The screenshot shows an XDC file with the following structure:

| Name of external FPGA pin | Electrical Standard used | The name to match the code input/output/... |
|--|------------------------------------|---|
| ## Clock signal | | |
| <code>{ PACKAGE_PIN W5 }</code> | <code>IOSTANDARD LVCMOS33 }</code> | <code>[get_ports clk]</code> |
| <code>create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports clk]</code> | | |
| ## Switches | | |
| <code>{ PACKAGE_PIN V17 }</code> | <code>IOSTANDARD LVCMOS33 }</code> | <code>[get_ports {sw[0]}]</code> |
| <code>{ PACKAGE_PIN V16 }</code> | <code>IOSTANDARD LVCMOS33 }</code> | <code>[get_ports {sw[1]}]</code> |
| <code>{ PACKAGE_PIN W16 }</code> | <code>IOSTANDARD LVCMOS33 }</code> | <code>[get_ports {sw[2]}]</code> |
| <code>{ PACKAGE_PIN W17 }</code> | <code>IOSTANDARD LVCMOS33 }</code> | <code>[get_ports {sw[3]}]</code> |
| <code>{ PACKAGE_PIN W15 }</code> | <code>IOSTANDARD LVCMOS33 }</code> | <code>[get_ports {sw[4]}]</code> |
| <code>{ PACKAGE_PIN V15 }</code> | <code>IOSTANDARD LVCMOS33 }</code> | <code>[get_ports {sw[5]}]</code> |
| <code>{ PACKAGE_PIN W14 }</code> | <code>IOSTANDARD LVCMOS33 }</code> | <code>[get_ports {sw[6]}]</code> |
| <code>{ PACKAGE_PIN W13 }</code> | <code>IOSTANDARD LVCMOS33 }</code> | <code>[get_ports {sw[7]}]</code> |
| <code>{ PACKAGE_PIN V2 }</code> | <code>IOSTANDARD LVCMOS33 }</code> | <code>[get_ports {sw[8]}]</code> |
| <code>{ PACKAGE_PIN T3 }</code> | <code>IOSTANDARD LVCMOS33 }</code> | <code>[get_ports {sw[9]}]</code> |
| <code>{ PACKAGE_PIN T2 }</code> | <code>IOSTANDARD LVCMOS33 }</code> | <code>[get_ports {sw[10]}]</code> |
| <code>{ PACKAGE_PIN R3 }</code> | <code>IOSTANDARD LVCMOS33 }</code> | <code>[get_ports {sw[11]}]</code> |
| <code>{ PACKAGE_PIN W2 }</code> | <code>IOSTANDARD LVCMOS33 }</code> | <code>[get_ports {sw[12]}]</code> |
| <code>{ PACKAGE_PIN U1 }</code> | <code>IOSTANDARD LVCMOS33 }</code> | <code>[get_ports {sw[13]}]</code> |
| <code>{ PACKAGE_PIN T1 }</code> | <code>IOSTANDARD LVCMOS33 }</code> | <code>[get_ports {sw[14]}]</code> |
| <code>{ PACKAGE_PIN R2 }</code> | <code>IOSTANDARD LVCMOS33 }</code> | <code>[get_ports {sw[15]}]</code> |
| ## LEDs | | |
| <code>{ PACKAGE_PIN U16 }</code> | <code>IOSTANDARD LVCMOS33 }</code> | <code>[get_ports {led[0]}]</code> |
| <code>{ PACKAGE_PIN E19 }</code> | <code>IOSTANDARD LVCMOS33 }</code> | <code>[get_ports {led[1]}]</code> |

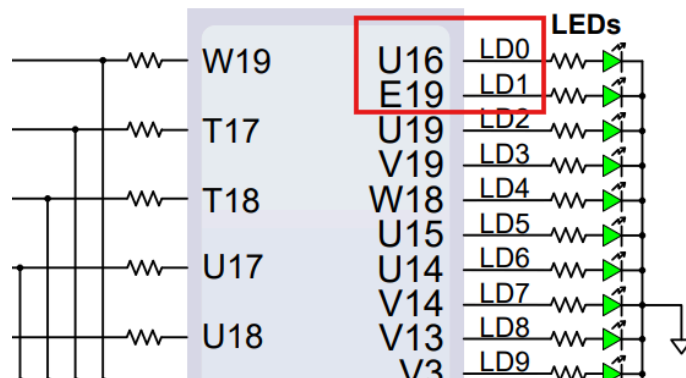
A red arrow points from the `[get_ports clk]` field in the XDC file to a Verilog module signature:

```
module moduleName (  
    input clk,  
    input [3:0] sw,  
    output [1:0] led  
);
```

The names of the input/output in top-level module must match the chosen pins in the XDC file

To find the detail of all pins in the Basys 3 board, you can read in the [reference manual](#).

For example, if we want to use 2 LEDs, we must find the pins that connect to the LEDs of the board, which are U16 and E19.



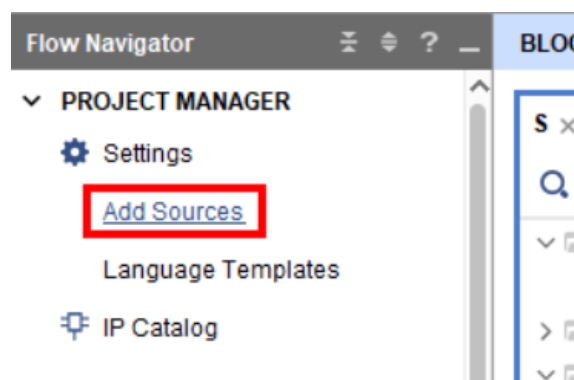
If the output port for LEDs in top level module is output [1:0] led, the constraints which must be specified in XDC file would be:

```
set_property -dict { PACKAGE_PIN U16  IOSTANDARD LVCMOS33 } [get_ports {led[0]}]

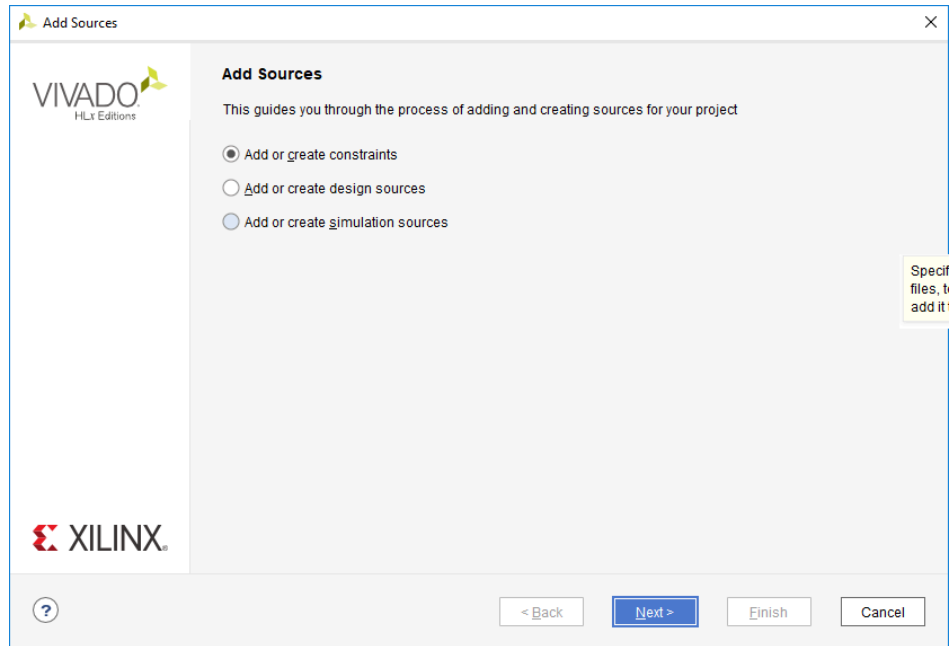
set_property -dict { PACKAGE_PIN E19  IOSTANDARD LVCMOS33 } [get_ports {led[1]}]
```

Importing XDC Files

- In the Project Manager section of the Flow Navigator pane, click the **Add Sources** button.



- On the first screen, select **Add or create constraints**. Click **Next** to continue.



- In the next screen, make sure that the constraint set specified (the one that the master XDC will be added to) is set to `constrs_1`, and that it is the active set. Click the **Add Files** button. Then select the XDC file we want to import.

