

Habib University



Dhanani School of Science and Engineering

Computer Architecture
(CS-330 / EE-471)

Lab Manual

Table of Contents

ABOUT THE LAB MANUAL	1
ABOUT THE LAB EXERCISES.....	3
CONVENTIONS	5
LAB 01 – INTRODUCTION	7
Objectives	7
a. Background.....	8
i. Categories of Logic Devices.....	8
ii. Evolution of Digital Designing using HDLs	9
iii. The importance of HDL	9
iv. Selection of HDL	10
b. Introduction to Verilog HDL	11
i. Lexical Elements.....	11
ii. Number representation.....	12
iii. Data Types	13
iv. Modules	14
v. Ports	14
vi. Instances	16
c. Hands-on Verilog HDL	17
i. Ripple Carry Counter	18
ii. T-flipflop	19
iii. D-flipflop	19
d. Functional Verification	20
i. Creating Testbench/Stimulus.....	20
ii. Verification in ModelSim®	23
Step-by-Step guide of functional verification in ModelSim®	23
Automated functional verification in ModelSim®	29
Exercise.....	31
BIBLIOGRAPHY	32

About the lab manual

This lab manual has been created with the help of practical experiments, several supporting documents and presentations listed in the Bibliography section.

The creation process of this manual was started during the summer 2018 by Dr. Hasan Baig, and it is continuously being updated during the Fall 2018 offering of the course “Computer Architecture”.

For questions, comments, suggestions, advices or corrections in this lab manual, please contact Dr. Hasan Baig at the following email address: hasan.baig@sse.habib.edu.pk.



About the lab exercises

These laboratory exercises have been designed to get the students acquainted with the hardware design skills. You will learn how to design hardware using hardware description language (HDL); how to simulate your design; and how to test it on a reconfigurable chip. Once you get familiar with the design flow, you will be required to develop processor peripherals in the following labs. A brief summary of all the lab exercises are given below.

In **Lab 1**, you will be introduced to the programmable logic and the Verilog HDL. Furthermore, you will learn how to design a simple hardware and verify its functional behavior using a professional simulation tool, named *ModelSim®*.

In **Lab 2a**, a hardware synthesis flow is discussed targeting the Xilinx FPGA technology. Furthermore, you will get to run your designed hardware on actual FPGA chip.

In **Lab 2b**, you will learn how to integrate the ready-made module (UART) with your custom design. Also, in this session, you will use desktop-based software, designed specifically for this course, to observe the output of on-chip hardware.

In **Lab 3**, you will be developing some intermediate modules of a processor which will be required in next labs. In particular, you will develop a multiplexer, and a decoder and perform simulation.

In **Lab 4**, In this lab, you will develop a Register File for a processor, and will simulate its behavior in ModelSim.

In **Lab 5**, You will develop a RISC arithmetic and logic unit and verify its functionality using simulation.

In **Lab 6**, you will learn how to use BRAM module in FPGA in order to implement the program memory of a processor. We will also implement a datapath for instruction fetch followed by functional verification.

In **Lab 7**, you will develop a single cycle RISC processor for R-type instructions and perform its behavioral simulation.

In **Lab 8**, you will design and test components for RISC processor that can handle branch instructions.

In **Lab 9**, you will design and test components for RISC processor that can handle memory reference instructions such as load and store.

In **Lab 10**, you will integrate the previously designed modules to form a single datapath for executing any type of instructions.

In **Lab 11**, you will design a control unit of RISC processor and then integrate it with the previously developed complete datapath.



Conventions

The following conventions appear in this lab manual.



This icon denotes a “pre-lab exercise”, which a student should complete before coming into the respective lab.



This icon denotes a “lab exercise”, which a student should complete during the lab hours.



This icon denotes a “post-lab exercise”, which a student should complete outside the lab hours.



This icon indicates the expected time (in minutes) to complete the specific exercise.



This icon denotes a tip, which notifies you to advisory information.



This icon denotes an alert, which notifies you to important information.

Bold or
Italic

The text written in this font is used specifically for the syntax of HDL.

bold

Bold text denotes items that you must select or click or enter the value in the software, such as open file option or running the simulation button or entering the command in the transcript window. The bold text is also used to refer to the specific options in the software tools.

italic

Italic text denotes the name of a folder or a file path.









bold and italic

Bold and italic text denotes the name of a file.

Lab 01 – Introduction

Objectives

This lab consists of four different sections. The description of each section is given below:

Section	
a) <u>Background</u>  The aim of this section is to discuss: i. The logic devices ii. The evolution of digital designing using HDL iii. The importance and selection of HDLs	10
b) <u>Introduction to Verilog HDL</u>   In this section, we will study the basic syntax of Verilog HDL	20
c) <u>Hands-on Verilog HDL</u>  In this part, we will do some practical examples in Verilog HDL. We will use a text editor named “Programmers Notepad” to write HDL codes.	60
d) <u>Functional verification</u>  In this part, we will learn to simulate and verify the functionality of a hardware developed, using HDL, in the previous part. A tool named ModelSim® will be used for functional verification.	40
<u>Exercise</u>   Small exercise to practice either at home or in the lab. If you have ModelSim running on your own PC, you can do the exercise at home.	20





a. Background

Three basic kinds of devices available in today's digital era are:

1. Memory devices – store random information as the contents of a spreadsheet or database.
2. Microprocessors – execute various software instructions to perform a wide variety of tasks.
3. Logic devices – provide specific functions, including device-to-device interfacing, data communication, signal processing, data display, timing and controlling operations, and almost every other function a system must perform.

i. Categories of Logic Devices

Logic devices can be classified into two broad categories – fixed and programmable. The circuits in fixed logic devices are permanent and can't be reprogrammed once after manufacturing, e.g. Application Specific ICs (ASICs). With fixed logic devices, the time required to go from design, to prototypes, to a final manufacturing run can take from several months to more than a year, depending on the complexity of the device. If the device does not work properly, or if the requirements change, a new design must be developed.

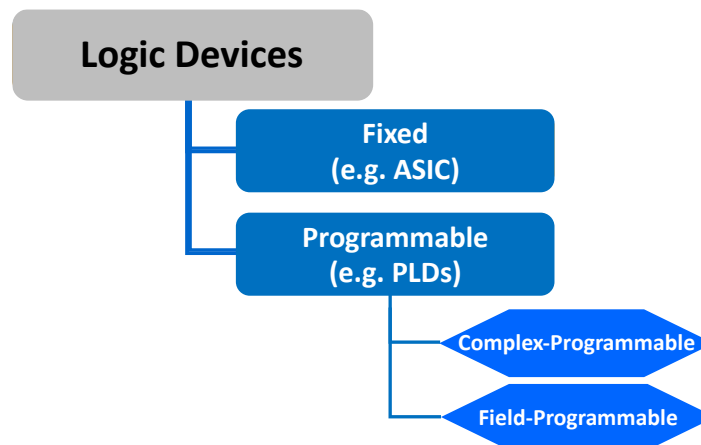


Figure 1.1. Categories of logic devices.

Whereas, programmable logic devices offer customers a wide range of logic capacity and functionality that can be reconfigured whenever required. For PLDs, designers use inexpensive software tools to quickly develop, simulate, and test their designs. Then, a design can be quickly programmed into a device, and immediately tested in a live circuit. The PLD that is used for this prototyping is the exact same PLD that will be used in the chip for the final product such as iPhones, USB flash drives etc. Major advantage of using PLDs is that designer can change the circuitry, during the design phase, as often as it requires until it meets the required functionality.



The two major types of programmable logic devices are field programmable gate arrays (FPGAs) and complex programmable logic devices (CPLDs). FPGAs are fine-grained devices and offer the highest amount of logic density (up to 100,000s of logic blocks). They are RAM based and needs to be configured at each power-up. FPGAs also have special routing resources to implement arithmetic functions efficiently. CPLDs, on the other hand, are coarse-grained devices and offer much smaller amounts of logic - up to about 10,000 gates. They are EEPROM based and are active at power-up. Like FPGAs, CPLDs do not have such special routing resources. In these labs, we will be using an FPGA device to create our own processor.

ii. Evolution of Digital Designing using HDLs

For a long time, programming languages such as FORTRAN, Pascal, and C were being used to describe computer programs that were sequential in nature. Similarly, in the digital design field, designers felt the need for a standard language to describe digital circuits. Thus, Hardware Description Languages (HDLs) came into existence. HDLs allow the designers to model the concurrency of processes found in hardware elements. Hardware description languages such as Verilog HDL and VHDL (Very High Speed Integrated Circuit (VHSIC) HDL) became popular. Verilog HDL originated in 1983 at Gateway Design Automation. Later, VHDL was developed under contract from Defense Advanced Research Project Agency (DARPA). Both Verilog and VHDL simulators to simulate large digital circuits quickly gained acceptance from designers.

Even though HDLs were popular for logic verification, designers had to manually translate the HDL-based design into a schematic circuit with interconnections between gates. The advent of logic synthesis in the late 1980s changed the design methodology radically. Digital circuits could be described at a Register Transfer Level (RTL) by use of an HDL. Thus, the designer had to specify how the data flows between registers and how the design processes the data. The details of gates and the interconnections to implement the circuit were automatically extracted by logic synthesis tools from the RTL description.

Thus, logic synthesis pushed the HDLs into the forefront of digital design. Designers no longer had to manually place gates to build digital circuits. They could describe circuits at an abstract level in terms of functionality and dataflow by designing those circuits in HDLs. Logic synthesis tools would implement the specified functionality in terms of gates and gate interconnections.

HDLs also began to be used for system-level design. HDLs were used for simulation of system boards, interconnect buses, FPGAs (Field Programmable Gate Arrays), and PALs (Programmable Array Logic). A common approach is to design each IC chip, using an HDL, and then verify system functionality via simulation.

iii. The importance of HDL

HDLs have many advantages compared to traditional schematic-based design.





- Designs can be described at an abstract level by use of HDLs. Designers can write their RTL description without choosing a specific fabrication technology. Logic synthesis tools can automatically convert the design to any fabrication technology. If a new technology emerges, designers do not need to redesign their circuits. They simply input the RTL description to the logic synthesis tool and create a new gate level netlist, using the new fabrication technology. The logic synthesis tool will optimize the circuit in area and timing for the new technology.
- By describing designs in HDLs, functional verification of the design can be done in the design cycle. Since designers work at the RTL, they can optimize and modify the RTL description until it meets the desired functionality. Most design bugs are eliminated at this point. This cuts down design cycle time significantly because the probability of hitting a functional bug at a later time in the gate-level netlist or physical layout is minimized.
- Designing with HDLs is analogous to computer programming. A textual description with comments is an easier way to develop and debug circuits. This also provides a concise representation of the design compared to gate level schematics. Gate-level schematics are almost incomprehensible for very complex designs.

HDLs are most certainly a trend of the future. With rapidly increasing complexities of digital circuits and increasingly sophisticated CAD tools, HDLs will probably be the only method for large digital designs. No digital circuit designer can afford to ignore HDL-based design.

iv. *Selection of HDL*

There are two major HDLs – Verilog and VHDL. One should consider the following points while choosing HDL.

- “Ease of Learning” which relates to how easy it is to learn the language without prior experience with HDLs.
- “Ease of Use” means once the first-time user has learned the language, how easy will it be to use the language for their specific design requirements.
- “Adaptability” is another important factor i.e., how the HDL can integrate into the current design environment and the existing design philosophy.

A wise guy is the one who takes the advantage of “other’s” experience to enhance his experience. Following are some quotes (related to choosing HDL) taken from the EE Times: the global electronic engineering community.

1.*I would really encourage any new HDL designer to choose Verilog rather than VHDL, since it is much easier to learn, use and eventually master.*
Scott C. Petler, Next Level Communications, Inc.





2. *In a previous life, I worked as an onsite application engineer for an ASIC vendor. The customer that I supported was developing 17 ASIC's for a large program. The customer chose to develop some of the designs in VHDL and others in Verilog. Designs done in Verilog were, without fail, completed faster than those done in VHDL.*

ASIC Foundry Engineer, Anonymous

3. *I have spent most of my design life (last 4 years) working on VHDL designs. Recently, I have been forced into the Verilog camp by a vendor. My initial concerns that Verilog would not have the functionality that I needed have been proven wrong. Verilog does what I need better - and the simulators are faster than VHDL simulators..... The behavioral compilers, not VHDL, make the most sense for doing even more sophisticated design work. Please don't make me go back to VHDL!*

Robert Rust

Hewlett Packard Boise Printer Division

b. Introduction to Verilog HDL

This section briefly describes the syntax of Verilog HDL and covers some basic lexical elements to write a Verilog HDL code.

i. Lexical Elements

The basic lexical conventions used by Verilog HDL are similar to those in the C programming language. Verilog HDL is a case-sensitive language. All keywords are in lowercase.

Here we will only discuss some of the lexical conventions that are new to Verilog. For a detailed reference, Verilog Language Reference manual or any other reference on Verilog may be consulted.

Identifier – An *identifier* gives a unique name to an object, such as `counter`, `seven_segment`, `el2`, etc. It is composed of letters, digits, the underscore character (`_`), and the dollar sign (`$`). `$` is usually used with a system task or function.



The first character of an identifier must be a letter or underscore.



Verilog is a *case-sensitive language*. Thus, `data-bus`, `Data-bus`, and `DATA_BUS` refer to three different objects. To avoid confusion, we should refrain from using the case to create different identifiers.

Keywords – *Keywords* are predefined identifiers that are used to describe language constructs. For example, **module**, **wire**, **not**, etc.





White space – White space, which includes the space, tab, and newline characters, is used to separate identifiers and can be used freely in the Verilog code. We can use proper white spaces to format the code and make it more readable.

Comments – A comment is just for documentation purposes and will be ignored by a compiler. Verilog has two forms of comments. A one-line comment starts with `//`, as:

```
// This is a comment
```

A multiple-line comment is encapsulated between `/*` and `*/`, as shown below:

```
/* This is comment line 1.
   This is comment line 2.
   This is comment line 3. */
```

ii. Number representation

There are two types of number specification in Verilog: sized and unsized.

Sized numbers

Sized numbers are represented as

```
[sign][size] '[base format] [number]
```

[sign] is written in the case of signed numbers. [size] is written only in decimal and specifies the number of bits in the number. Legal base formats are decimal ('d or 'D), hexadecimal ('h or 'H), binary ('b or 'B) and octal ('o or 'O). The number is specified as consecutive digits from 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f. Only a subset of these digits is legal for a particular base. Uppercase letters are legal for number specification.

```
4'b1111    // This is a 4-bit binary number
12'habc     // This is a 12-bit hexadecimal number
16'd255     // This is a 16-bit decimal number
```

Unsized numbers

Numbers that are specified without a [base format] specification are decimal numbers by default. Numbers that are written without a <size> specification have a default number of bits that is simulator- and machine specific (must be at least 32).

```
23456      // This is a 32-bit decimal number by default
'hc3       // This is a 32-bit hexadecimal number
'o21       // This is a 32-bit octal number
```

X or Z values

Verilog has two symbols for unknown and high impedance values. These values are very important for modeling real circuits. An unknown value is denoted by **x**. A high impedance value is denoted by **z**.





```

12'h13x    // This is a 12-bit hex number; 4 least significant
           // bits unknown
6'hx       // This is a 6-bit hex number
32'bz      // This is a 32-bit high impedance number

```

An **x** or a **z** sets four bits for a number in the hexadecimal base, three bits for a number in the octal base, and one bit for a number in the binary base. If the most significant bit of a number is 0, x, or z, the number is automatically extended to fill the most significant bits, respectively, with 0, x, or z. This makes it easy to assign x or z to whole vector. If the most significant digit is 1, then it is also zero extended.

Negative numbers

Putting a minus sign before the size for a constant number can specify negative numbers. Size constants are always positive. It is illegal to have a minus sign between <base format> and <number>.

```

-6'd3      //8-bit negative number stored as 2's complement of 3
4'd-2      // illegal specification

```

iii. Data Types

Four basic values are used in most data types:

0: for "logic 0, or a false condition
 1: for "logic 1", or a true condition
z: High impedance, floating state
x: for an unknown value

The **z** value corresponds to the output of a tri-state buffer. The **x** value is usually used in modeling and simulation, representing a value that is not 0, 1, or z, such as an uninitialized input or output conflict.

Verilog has two main groups of data types: `net` and `register`

Nets

Nets represent connections between hardware elements. Just as in real circuits, nets have values continuously driven onto them by the outputs of devices that they are connected to.

Nets are declared primarily with the keyword **wire**. Nets are one-bit values by default unless they are declared explicitly as vectors.

```

wire p0, p1;           // two 1-bit signals
wire [7:0] data1, data2; // 8-bit data
wire [31:0] addr;      // 32-bit address
wire [0:7] revers-data; //ascending index should be avoided

```





While the index range can be either descending (as in [7:0]) or ascending (as in [0:7]), the former is preferred since the leftmost position (i.e., 7) corresponds to the MSB of a binary number.



It is possible to address bits or parts of a vector.

For example,

`data1[3]` refers to the bit 3 of a wire `data1` declared above.

`addr[2:0]` three least significant bits of a vector `addr`

The term **wire** and **net** are often used interchangeably. The default value of a net is **z**. Nets get the output value of their drivers. If a net has no driver, it gets the value **z**.

Registers

Registers represent storage data elements. Registers retain value until another value is placed onto them. Do not confuse the term registers in Verilog with hardware registers built from edge triggered flip-flops in real circuits. In Verilog, the term register merely means a variable that can hold a value. Unlike a net, a register does not need a driver. Verilog registers do not need a clock as hardware registers do. Values of registers can be changed anytime in a simulation by assigning a new value to the register.

Register data types are commonly declared by the key word `reg`. The default value for a `reg` data type is **x**. An example of how registers are declared and used is shown below:

```
reg reset;    // declare a variable that can hold its value
```

iv. Modules

Verilog provides the concept of a module. A module is the basic building block in Verilog. A module can be an element or a collection of lower-level design blocks. A module provides the necessary functionality to the higher-level blocks through its port interface (inputs and outputs) but hides the internal implementation. This allows the designer to modify module internals without affecting the rest of the design.

In Verilog, a module is declared by the keyword **module**. A corresponding keyword **endmodule** must appear at the end of the module definition. Each module must have a *module_name*, which is the identifier for the module, and a *module_terminal_list*, which describes the input and output terminals of the module.

```
module      <module_name> (<module_terminal_list>);
. . .
<module internals>
. . .
endmodule
```

v. Ports

Ports provide the interface by which modules can communicate with its environment. For example, the *input/output* pins of an IC chip are its ports. The environment can interact with the





module only through its ports. The internals of the module are not visible to the environment. This provides a very powerful flexibility to the designer. The internals of the module can be changed without affecting the environment as long as the interface is not modified. Ports are also referred to as *terminals*.



A module definition contains as optional list of ports. If the module does not exchange any signals with the environment, there are no ports in the list.

Port Declaration

All ports in the list of ports must be declared in the module. Ports can be declared as follows

Verilog Keyword	Type of Port
input	Input port
output	Output port
Inout	Bi-directional port

Each port in the port list is defined as input, output, or inout, based on the direction of the port signal. Thus, for the example of the `fulladd4`, the port declarations will be as shown below:

```
module fulladd4 (sum, c_out, a, b, c_in);

    // Begin port declarations section
    output [3:0] sum;
    output c_out;

    input [3:0] a, b;
    input c_in;
    // End port declarations section
    ...
    <module internals>
    ...
endmodule
```

Note that all port declarations are implicitly declared as **wire** in Verilog. Thus, if port is intended to be a **wire**, it is sufficient to declare it as **output**, **input**, or **inout**. Input or inout ports as normally declared as **wire**. However, if output ports hold their value, they must be declared as **reg**. For example, consider a module for a D-type flip-flop:

```
module DFF (q, d, clk, reset);
    output reg q;    // Output port q holds value; therefore it
                    // is declared as reg
    input d, clk, reset;
    ...
    ...
endmodule
```





Ports of the type **input** cannot be declared as **reg**, because **reg** variables store values and **input** ports should not store values but simply reflect the changes in the external signals they are connected to.

vi. Instances

A module provides a template from which you can create actual objects. When a module is invoked, Verilog creates a unique object from the template. Each object has its own name, variables, parameters and I/O interfaces. The process of creating objects from a module template is called *instantiation*, and the objects are called *instances*. In the example below, which shows the module for a 4-bit ripple carry counter, four instances from the T-flipflop template are created. The internals of the T_FF module are not shown.

```
// Define the top-level module called ripple carry counter.
// It instantiates 4 T-flipflops.

module ripple_carry_counter (q, clk, reset);

output [3:0] q;
input clk, reset;

// Four instances of the module T_FF are created. Each has a
// unique name. Each instance is passed a set of signals.
// Notice that each instance is a copy of the module T_FF

T_FF tff0 (.q(q[0]), .c(clk), .reset(reset));
T_FF tff1 (.q(q[1]), .c(q[0]), .reset(reset));
T_FF tff2 (.q(q[2]), .c(q[1]), .reset(reset));
T_FF tff3 (.q(q[3]), .c(q[2]), .reset(reset));

endmodule
```



The port name of a calling module (the module being instantiated) comes first. In above example, the port **c** of a **T_FF tff0** module is connected to the port **clk** of a **ripple_carry_counter**.



For further details about Verilog HDL, go through the references [1][2].



c. Hands-on Verilog HDL

In this section, we will develop an example hardware of a 4-bit ripple carry counter using the concepts learnt in previous section. The top-level diagram of a 4-bit ripple carry counter is shown in the Figure 1.2. It is designed with four negative edge-triggered T-flipflops.

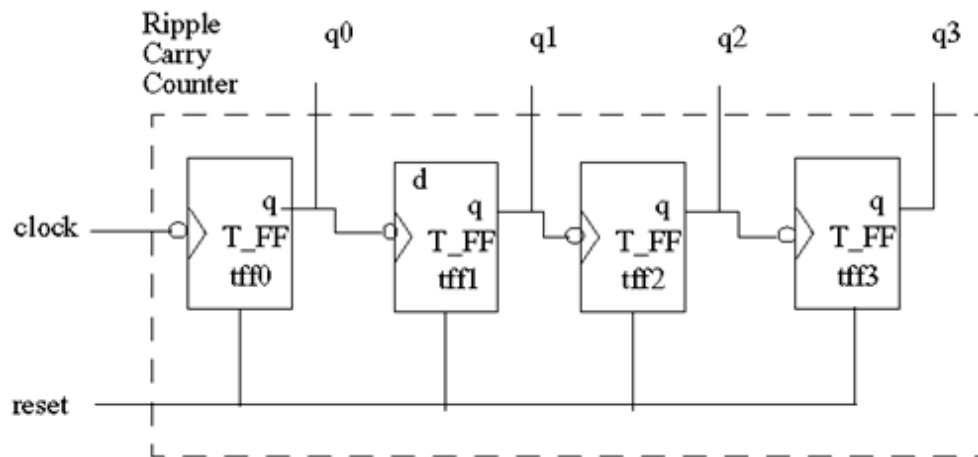


Figure 1.2. Top-level diagram of a ripple carry counter.

Each of these above T-flipflops can be made from a negative edge-triggered D-flipflop and an inverter (assuming that a q' output is not available). The implementation of T_FF using D_FF and an inverter is shown in Figure 1.3.

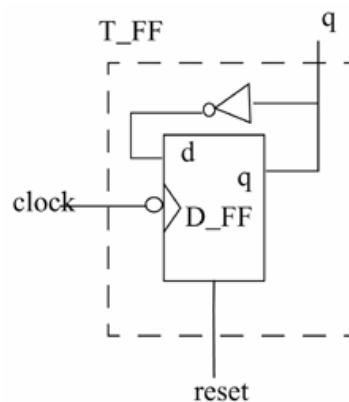


Figure 1.3. Implementation of a T_FF using D_FF and inverter.



Open Programmers notepad and start designing the above modules as shown in the next sub-section.

For this lab, we will be working in the following directory.



C:\CA_Labs\Lab01\design

Create the above directory before proceeding ahead. All of the design modules we are going to develop will be stored in it.



i. Ripple Carry Counter

Using the top-down design approach, we will first write the verilog description of a top-level module, which is a 4-bit ripple carry counter. The verilog code of ripple carry counter is shown in Figure 1.4 below.

```

1  module ripple_carry_counter
2  (
3      input clk, reset,
4      output [3:0] q
5  );
6
7      //Instantiate T-FF modules
8
9      T_FF tff0
10     (
11         .c(clk),
12         .r(reset),
13         .q(q[0])
14     );
15
16     T_FF tff1
17     (
18         .c(q[0]),
19         .r(reset),
20         .q(q[1])
21     );
22
23     T_FF tff2
24     (
25         .c(q[1]),
26         .r(reset),
27         .q(q[2])
28     );
29
30     T_FF tff3
31     (
32         .c(q[2]),
33         .r(reset),
34         .q(q[3])
35     );
36 endmodule

```

Figure 1.4. Verilog description of a top-level module named `ripple_carry_counter`.



The module name should match the file name as shown in the figure above.

In `ripple_carry_counter` module, shown above, four instances of the module `T_FF` (`tff0` to `tff3`) are used. Since the port name of a module being instantiated comes first, therefore it is clear from the above instantiation that the `T_FF` module has three port names `c`, `r`, and `q`.



The syntax of a not gate in verilog is:

`not <instance_name> (output, input)`

Since there could be multiple not gates in the same module, therefore it is mandatory to define the different instance name for each instantiation.



ii. T-flipflop

We now have to define the internals of the module `T_FF`. The verilog description of a `T_FF` module is shown in Figure 1.5. This module contains an instance of a `D_FF` module, having four ports. The `T_FF` module also contains a NOT gate which connects the output of a `D_FF` i.e. `q`, with its input i.e. `d`. This implementation is already shown in Figure 1.3.

```

1  module T_FF
2  (= (
3      input c, r,
4      output q
5  );
6      wire d;
7
8      //Instantiate D_FF
9      D_FF dff0
10     (= (
11         .c(c),
12         .r(r),
13         .q(q),
14         .d(d)
15     );
16
17     not n1(d,q);
18 endmodule

```

Figure 1.5. Verilog description of a `T_FF` module.

iii. D-flipflop

We are now left with designing the leaf module which is D-flipflop. The verilog description of a `D_FF` module is shown in Figure 1.6. This module contains an always block which is activated either at the `posedge` (positive edge) of `r` (reset signal) or at the `negedge` (negative edge) of `c` (clock signal). When a reset signal is triggered, the output of `D_FF` resets to 0. When reset is disserted, the output `q` retains the value of input `d` on each negative edge of a clock signal.



```

1  module D_FF
2  (
3      input c, r,
4      input d,
5      output reg q
6  );
7
8      always @ (posedge r or negedge c)
9      begin
10         if (r)
11             q <= 1'b0;
12         else
13             q <= d;
14         end
15
16 endmodule

```

Figure 1.6. Verilog description of a D_FF module.



Note that the data type of the output port q is defined as `reg` only in `D_FF` module and not in any other top-level modules. It is because the `D_FF` is the main source of generating the output q , therefore it should be defined as `reg`. The generated output signal q then simply propagates to the top-level modules, therefore the data type of the output ports of these modules are simply `net`.

The design of 4-bit ripple carry counter module is complete. We now have to verify its functionality, which is demonstrated in the next section.

d. Functional Verification

Once a design block is completed, it must be tested. The functionality of a design block can be tested by applying stimulus and checking results. We call such a block the *stimulus* block. It is good practice to keep the stimulus and design block separate. The stimulus block can be written Verilog. A separate language is not required to describe stimulus. The stimulus block is also commonly called a *test bench*. Different test benches can be used to thoroughly test the design block.

i. Creating Testbench/Stimulus

Two styles of stimulus application are possible. In the first style, the stimulus block instantiates the design block and directly drives the signals in the design block. In Figure 1.7, the stimulus block becomes the top-level block. It manipulates the signals `clk` and `reset`, and checks and displays output signal q .

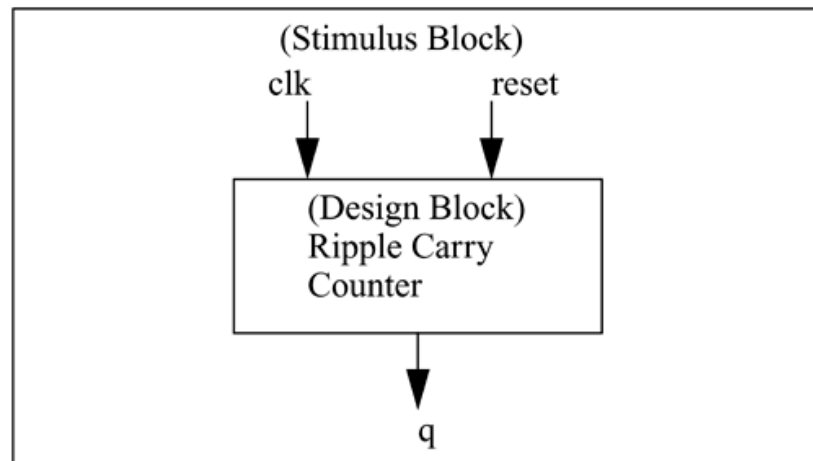


Figure 1.7. Stimulus block instantiates design block.

The second style of applying stimulus is to instantiate both the stimulus and design blocks in a top-level dummy module. The stimulus block interacts with the design block only through the interface. This style of applying stimulus is shown in Figure 1.8. The stimulus module drives the signal `d_clk` and `d_reset` in the design block. It also checks and displays signal `c_q`, which is connected to the signal `q` in the design block. The function of top-level block is simply to instantiate the design and stimulus blocks.

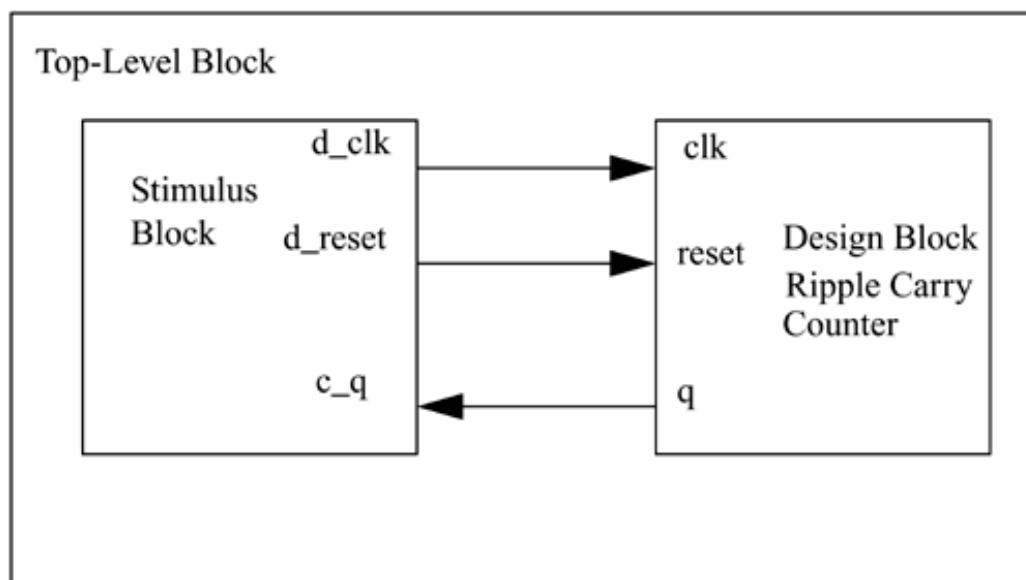


Figure 1.8. Stimulus and Design Blocks Instantiated in a Dummy Top-Level Module.

Either stimulus style can be used effectively. In this lab, we will use the stimulus block approach shown in Figure 1.7.



In stimulus block approach (Figure 1.7), stimulus block acts like an environment to provide inputs to the design block and observe its outputs. Therefore, the stimulus block does not require any I/O ports interface.

We must now write the description of a stimulus block to check if the design module `ripple_carry_counter` is functioning correctly. In this case, we must control the



signals `clk` and `reset` so that the regular function of the ripple carry counter and the reset mechanism are both tested.

The Verilog module of the testbench for testing `ripple_carry_counter` design is shown in Figure 1.9. It is clearly shown that the module `tb` in Figure 1.9 doesn't have any I/O ports. We only have internal connections declared as a `wire/reg`. Now the Design Under Verification (DUV) is instantiated and its ports are connected with the stimulus signals which, in our case, are `clk`, `reset` and `q`. Then the clock signal (`clk`) is initialized and generated with a time period of 10 units i.e. it toggles between 1 and 0 after each 5 units, so the time period is 10 units. The `initial` block (line 16) is used to assign the values at the beginning of simulation. This block is not synthesizable, and therefore only used for simulation purpose. The `always` block (line 19) enforce the clock signal to toggle continuously each after 5 time units.

```

1  module tb
2  (
3
4  );
5      reg clk;
6      reg reset;
7      wire [3:0] q;
8
9      ripple_carry_counter r1
10     (
11         .clk(clk),
12         .reset(reset),
13         .q(q)
14     );
15
16     initial
17     clk = 1'b0;
18
19     always
20     #5 clk = ~clk;
21
22     initial
23     begin
24         reset = 1'b1;
25         #15 reset = 1'b0;
26         #180 reset = 1'b1;
27     end
28
29     initial
30     $monitor("Time = ", $time, "---> Output = %d", q);
31 endmodule

```

Figure 1.9. Stimulus/testbench for testing `ripple_carry_counter` module.

Similarly, the `reset` signal is defined to be high initially; deasserted after 15 time units, and then asserted again after the total of 195 time units. `$monitor` block is used to print the results on the transcript window of Modelsim®.

Once the testbench/stimulus is generated, we are now ready to perform simulation in Modelsim®.





ii. Verification in ModelSim®

Modelsim® is a verification and simulation tool for Verilog, VHDL, System Verilog, System C, and mixed language designs. The simulation flow of ModelSim® is shown in Figure 1.10. This flow assumes that the design files have already been developed outside the ModelSim® environment. In our case, we have already developed design files using “Programmer’s Notepad”.

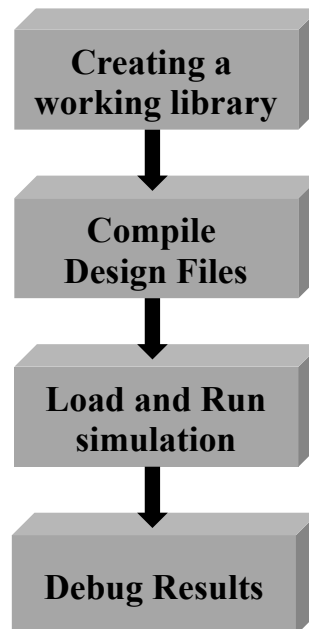


Figure 1.10. Simulation flow in ModelSim®.

In ModelSim, all the designs are compiled into a library. “*Work*” is the default library name for the compiler. We have to create a library (folder) named *work* in the directory we want to work in. i.e. in the Present Work Directory (PWD). After creating the working library, the design units can be compiled into it.

After compilation, the simulator has to be loaded with our design by invoking the simulator on a *top-level* module. Once the simulation is loaded, the design can be debugged using waveforms or printed results in the transcript window. All the steps of simulation are shown below in sequential manner.

Step-by-Step guide of functional verification in ModelSim®

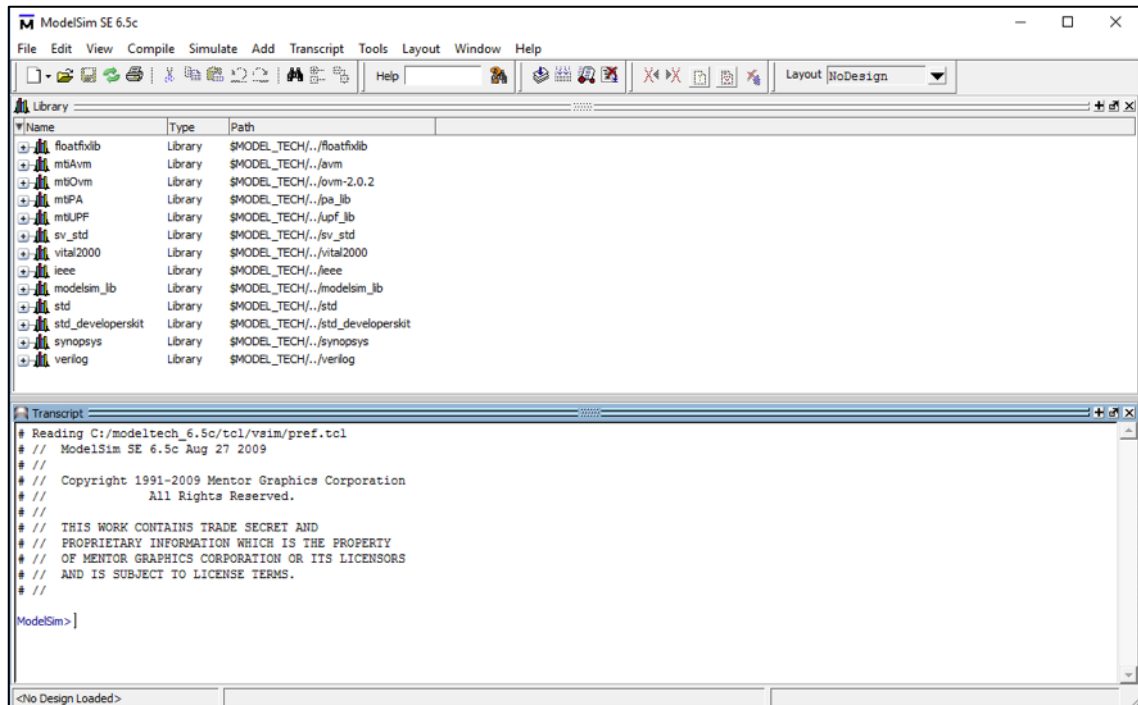
1. Launch ModelSim®

Launch Modelsim® by double clicking on the following **ModelSim SE 6.5c** icon on your desktop, as shown in the image below.



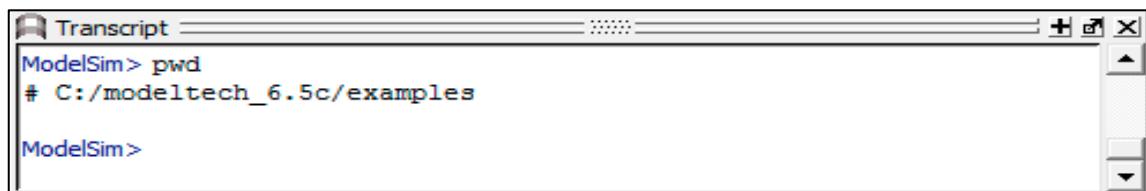


After double clicking on the above icon, the front interface of Modelsim® will appear as shown in the image below.



2. Present Work Directory

The default working directory of ModelSim® is *C:/modeltech_6.5c/examples*. Check the present work directory by typing **pwd** in the **Transcript** window, as shown in the image below.



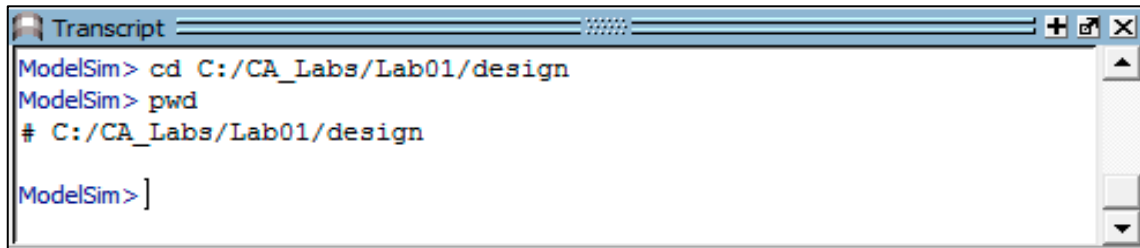
3. Change Directory

Now change the present work directory to the directory we have been working in till yet i.e. *C:\CA_Labs\Lab01\design*.



Modelsim requires forward slash (/) in the file path rather than a conventional backward slash (\).

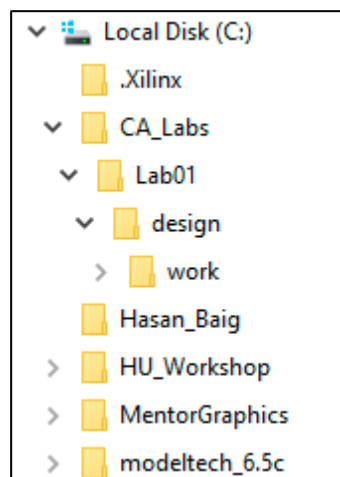
Type **cd C:/CA_Labs/Lab01/design** in the **Transcript** window to change the directory. Type **pwd** to verify that the present working directory has successfully been changed, as shown in the image below.



```
ModelSim> cd C:/CA_Labs/Lab01/design
ModelSim> pwd
# C:/CA_Labs/Lab01/design
ModelSim> ]
```

4. Create Work Library

Create *work* library by typing **vlib work** in the **Transcript** window. This will create a *work* folder in the present working directory. i.e. under *C:\CA_Labs\Lab01\design* directory.



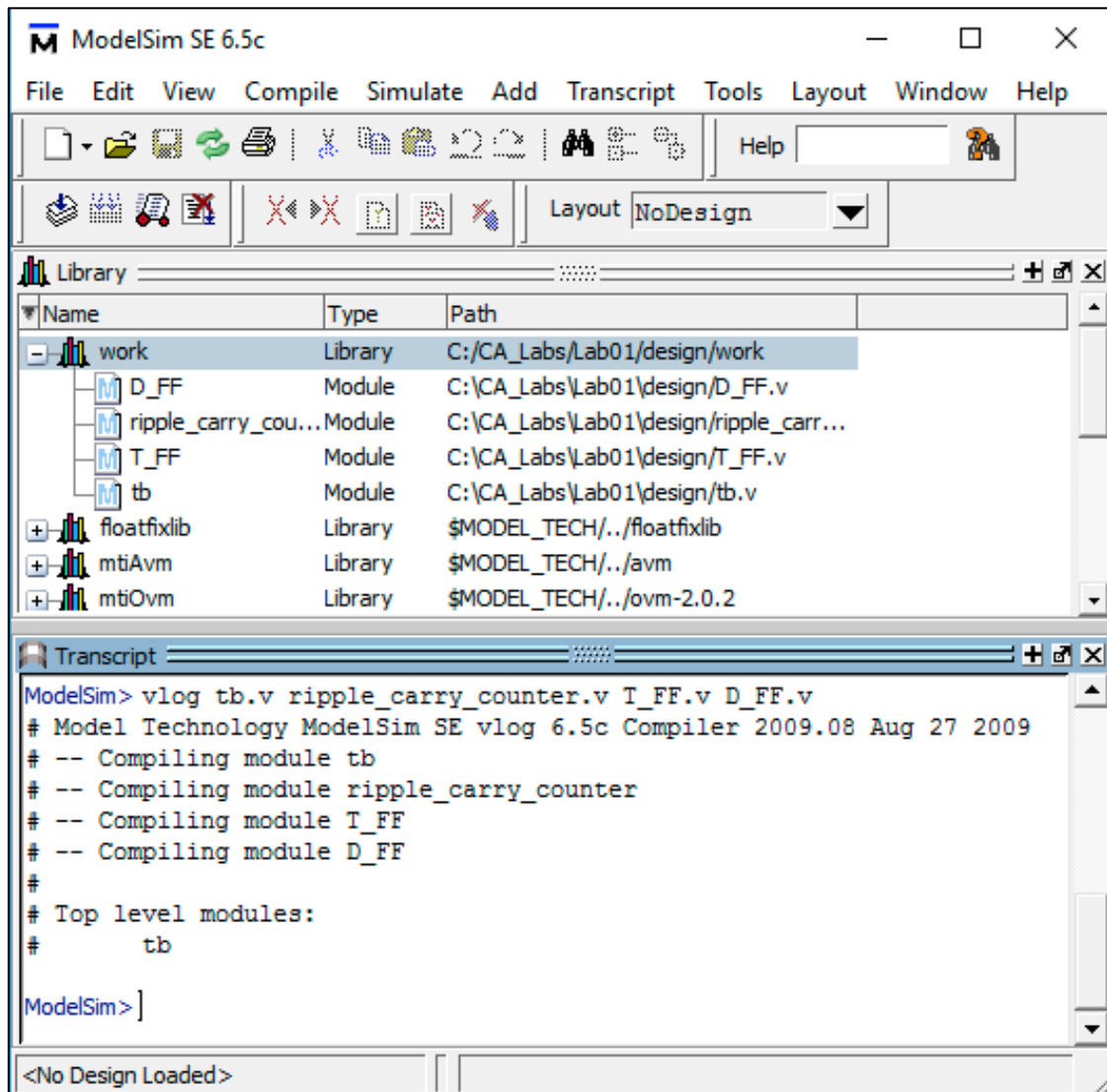
5. Compile Design Modules

To compile all the design files i.e. *tb.v*, *ripple_carry_counter.v*, *T_FF.v* and *D_FF.v*, type **vlog tb.v ripple_carry_counter.v T_FF.v D_FF.v** in the **Transcript** window. If all the files are compiled without any error, the following results will be displayed in the **Transcript** window (see image below), otherwise the errors will be highlighted in **red**.



The naming sequence of files in the compilation command does not matter.

When the design modules are compiled successfully, the file names will appear under *work* directory in the **Library** window of ModelSim® tool, as shown in the image below.

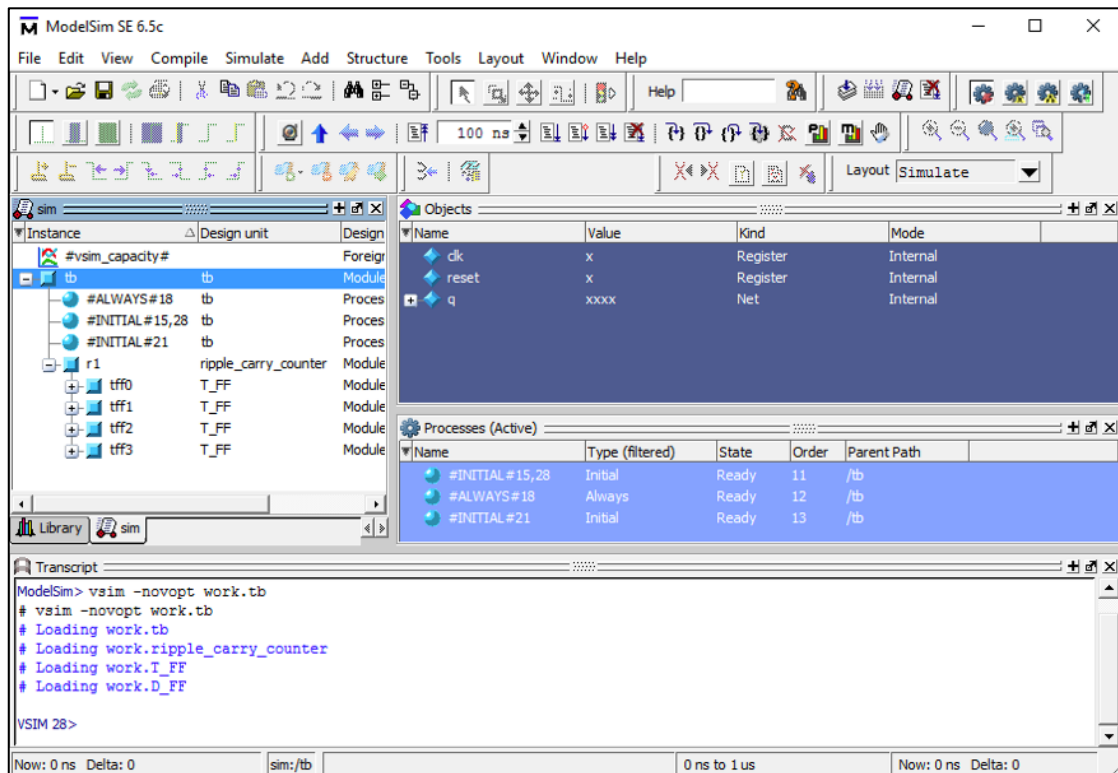


6. Loading Design in Simulator

Optimized designs simulate faster in ModelSim®. Optimization should be disabled to make the objects visible for debugging.

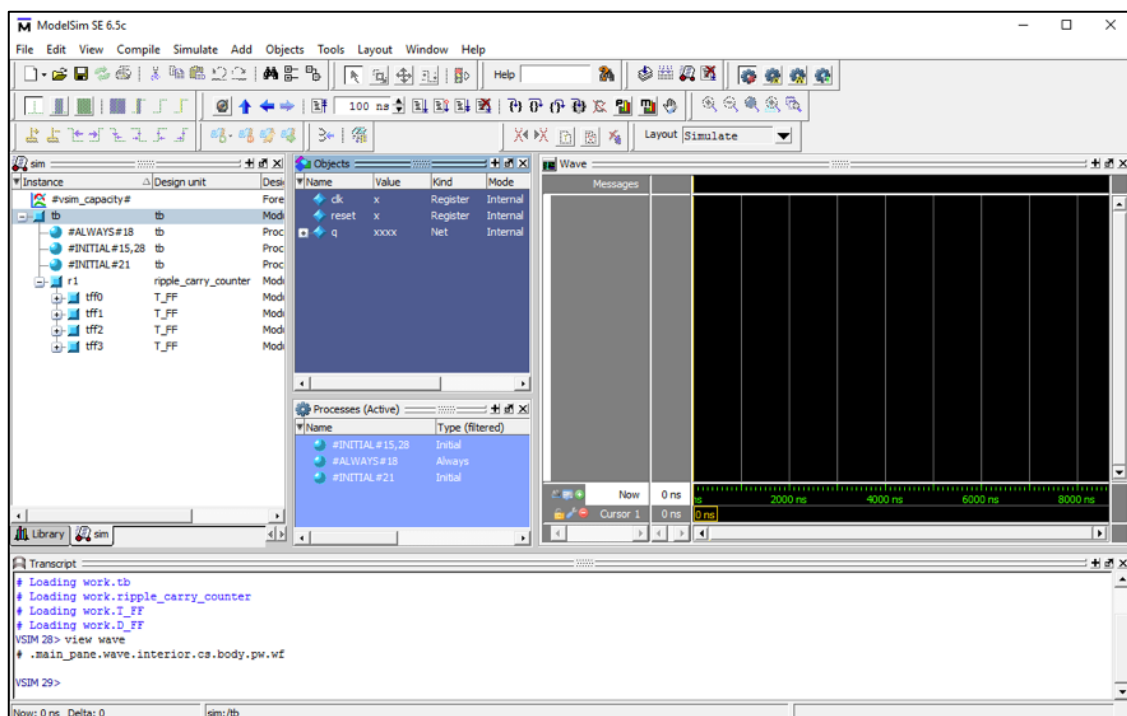
Type **`vsim -novopt work.tb`** in the **Transcript** window. A new window will appear that looks like the following image.

You can see the hierarchy of your design in the **sim** window (shown on the left side in the following figure). Each instance can be expanded down to see its details. When you click an instance, its variables are displayed in the **Objects** window.



7. Viewing and adding waves

Now we have to view the waveforms for debugging. Type **view wave** in the **Transcript** window which will open a new **Wave** window as shown in the following image.

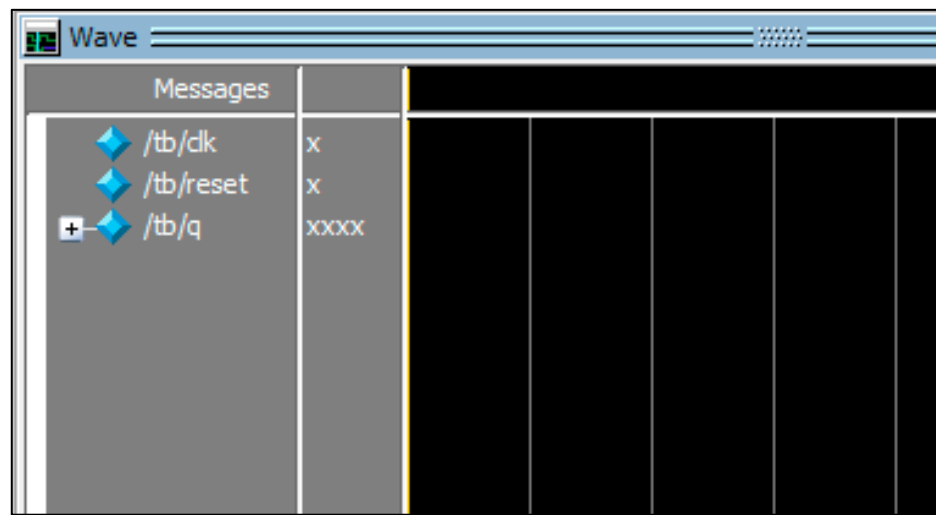




8. Adding signals to wave window

Add signals that you want to debug. If you prefer viewing all signals instead of some, you can go to **sim** window, select the top module (in this case, the top module is tb) and **Right click > Add > To Wave > All items in region**. All the signals will appear in the **Wave** window.

Here we'll add only `clk`, `reset` and `q` signals. In the **Objects** window, press and hold **Ctrl** Key and select all of the three signals. **Right click > Add > To Wave > Selected Signals**. The selected signals will appear in the **Wave** window as shown below.



9. Running the Simulation

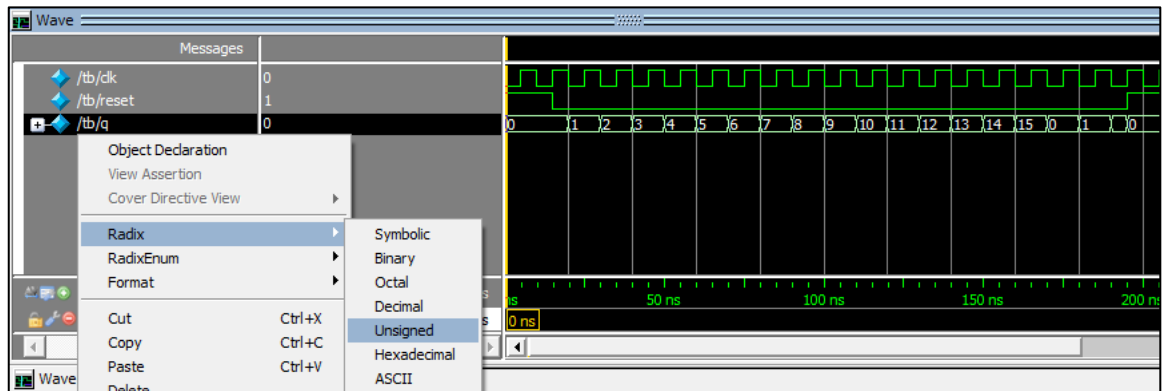
Run the simulation for desired length of time. For example, type **run 300ns** in the **Transcript** window to run the design modules for 300 ns.


We can see the results either on **Transcript** window or on **Wave** window. The outputs on the **Transcript** window will look like the following image. Here the results of the outputs are displayed for each clock cycle, because of the `$monitor` command we used in the `tb` module in Figure 1.9.

```
# Time = 0---> Output = 0
# Time = 20---> Output = 1
# Time = 30---> Output = 2
# Time = 40---> Output = 3
# Time = 50---> Output = 4
# Time = 60---> Output = 5
# Time = 70---> Output = 6
# Time = 80---> Output = 7
# Time = 90---> Output = 8
# Time = 100---> Output = 9
# Time = 110---> Output = 10
# Time = 120---> Output = 11
# Time = 130---> Output = 12
# Time = 140---> Output = 13
# Time = 150---> Output = 14
# Time = 160---> Output = 15
# Time = 170---> Output = 0
# Time = 180---> Output = 1
# Time = 190---> Output = 2
# Time = 195---> Output = 0
```



Also, the simulation results on the **Wave** window will look like the following image. Set the display of output signal “q” as shown in the figure below.



You can either use the Right click on the **Wave** window or you can use the menu buttons  to zoom in or zoom out in different regions.

Now in the simulation results, it can be observed that the `reset` signal was set high until a 15 ns. The output during this period is 0. When the `reset` signal is pulled down, the counter starts increment from the next subsequent negative-edge of the `clk` signal. It keeps counting until the value of 15 and then starts counting again from 0 (because it is a 4-bit counter). The counter value become 0 again at 195 ns, because the `reset` signal is asserted again. This is exactly how we expect the ripple carry counter to behave.

By this time, you should be able to compile and simulate your design to view the results. In next section we shall see how we can automate these steps to save our time in the debug process, where we need to simulate the design multiple times.

To quit simulation, type **quit -sim** in the **Transcript** window.

Automated functional verification in ModelSim®

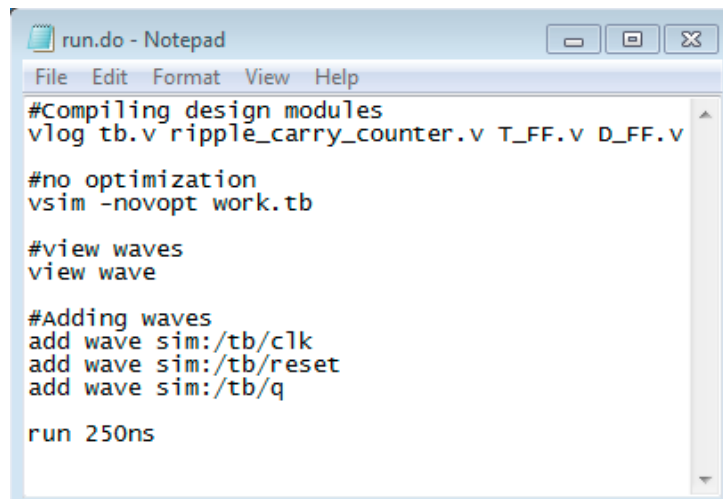
It is often desirable to do a small change in Verilog code and run the simulation again. To avoid doing the above-mentioned steps again we can automate these steps using the *script* file. To run the specific commands in sequence, we can write all the commands in a file and then use that file to run all the commands. For example, the set of commands studied above can be written in a separate file and that file needs to be run only once to run all those commands in sequence.

Create **run.do** file in the same working directory, i.e `C:\CA_Labs\Lab01\design`.



Create `run.txt` file first and rename its extension to `.do`.

Now add the following commands, in this file, that have already been studied in the abovementioned steps 5-to-8. `#` is used for commenting as shown in the image below.



```
run.do - Notepad
File Edit Format View Help
#Compiling design modules
vlog tb.v ripple_carry_counter.v T_FF.v D_FF.v

#no optimization
vsim -novopt work.tb

#view waves
view wave

#Adding waves
add wave sim:/tb/c1k
add wave sim:/tb/reset
add wave sim:/tb/q

run 250ns
```

Type **source run.do** or **do run.do** in the **Transcript** window to run all the commands sequentially to generate the simulation results.



Exercise

Create a folder named *Exercise* under `..\Lab01\design\` path and copy all of your design (.v) files of ripple carry counter module in it. Now work on the copied files to incorporate the following requirements.

1. Update the ripple carry counter to count until the value 31.
2. Make the ripple carry counter count at the positive edge of clock.

Run the simulation for 500 ns and verify the functionality of your design. Observe how the counting is different than the one developed before.

Bibliography

- [1]. Samir Palnitkar, *Verilog HDL: A guide to Digital Design and Synthesis*, Second Edition, Prentice Hall, 2003.
- [2]. Pong P. Chu, *FPGA Prototyping by Verilog Examples*, John Wiley & Sons Inc., 2008.
- [3]. Avnet, “Xilinx Spartan-6 LX9 Microboard User Guide”, RevD, April 2015.
- [4]. Bob Zeidman, *Introduction to CPLD and FPGA Design*, www.chalknet.com.
- [5]. Macro Platzner, *Evolution of Programmable Logic Devices (PLDs)*, Semester course “EECS 700” of Universität Paderborn, 2006-07.
- [6]. Xilinx Help, <http://www.xilinx.com/company/about/programmable.html>.
- [7]. <http://www.fpga4fun.com/FPGAinfo1.html>.
- [8]. Bill Fuchs, Simucad Inc., *Verilog HDL vs. VHDL*, A survey, 1995.
- [9]. John F. Wakerly, *Digital Design Principles and Practices*, Third Edition, Prentice Hall.
- [10]. Janick Bergeron, *Writing Testbenches: Functional Verification of HDL Models*, Kluwer Academic.
- [11]. Mentor Graphics, *Modelsim SE Tutorial*, Version 6.5c.
- [12]. Mentor Graphics, *Modelsim Commands*, Version 6.5c.
- [13]. Platform Studio, *Using a Modelsim Script File to Compile, Load, Stimulate, and Simulate a Design*, http://www.xilinx.com/itp/xilinx9/help/platform_studio/html/ps_p_sim_using_script_files_load_stim_sim_modelsim.htm.
- [14]. Mentor Graphics, *ModelSim SE User’s Manual*, Version 6.5c.
- [15]. XILINX, *ISE Design Suite Overview*, ISE Help (v 12.2), 2010.
- [16]. XILINX, *XST User Guide*, UG627 (v 11.3), 2009.
- [17]. Greg Gibeling, *Using Project Navigator*, A semester course “EECS150” of U.C Berkley, Fall 2004.
- [18]. Susan Lysecky, *XILINX ISE 8.1 Synthesis Tutorial*, A semester course “ECE 274” of University of Arizona, Spring 2007.
- [19]. XILINX, *PlanAhead User Guide*, UG632 (v12.2), 2010.
- [20]. XILINX, *ML505/ML506/ML507 Evaluation Platform User Guide*, UG347 (v3.1.1), 2009.
- [21]. XILINX, *Virtex-5 FPGA Packaging and Pinout Specification*, UG195 (v4.7), 2009.
- [22]. XILINX, *ML505/506/507 Overview and Setup*, Presentation slides, 2008.
- [23]. Prof. Gerald E. Sobelman, *Digital Design with Programmable Logic*, A semester course “EE 4301” of University of Minnesota, Summer 2010.
- [24]. Douglas J. Smith, *HDL Chip Design*, Doone Publications, 1997.
- [25]. Ben Cohen, *Real Chip Design and Verification Using Verilog and VHDL*, VhdlCohen Publilshing, 2002.
- [26]. REC FPGA Seminar IAP 1998, *Advanced Design Techniques, Optimizations, and Tricks*, Robotics and Electronics Cooperative FPGA Seminar IAP 1998.
- [27]. J. Bhasker, *Verilog HDL Synthesis – A Practical Primer*, Star Galaxy Publishing, 1998.
- [28]. Iqra University, “EEN-222 Computer Architecture”.
- [29]. Hasan Baig, “Jump-start to FPGA based Digital Designing and CAD Tools”, *workshop presentation*, 2010.
- [30]. UART, https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter.



Computer Architecture
EE-371L/CS-330L
Lab # 01

Name: _____ Student ID.: _____

Lab # 01 Marks Distribution:

	LR2=20	LR3=20	LR5=20	LR6=10	LR7=20
Task 1	10 points	-	-	-	10 points
Task 2	-	10 points	10 points	-	
Task 3	10 points	10 points	10 points	-	10 points
Total Marks	90 points				

Lab # 01 Marks Obtained:

	LR2=20	LR3=20	LR5=20	LR6=10	LR7=20
Task 1		-	-	-	
Task 2	-			-	-
Task 3					
Total Marks					

Lab Evaluation Assessment Rubric

#	Assessment Elements	(0<Level 1<=4)	(4< Level 2<=6)	(6< Level 3<=8)	(8< Level 4<=10)
LR1	Neat and Clean Circuit layout	Components are wired and didn't show neat and clean connections and minimal efforts shown	Components are wired with untidy connection and didn't show neat and clean connections	Few but not all Components are wired with neat and clean connections	Complete components are wired with neat and clean tight connections and task completed in due time
LR2	Program/code/simulation model/network model	Program/code/simulation model/network model is not well-documented (i.e. with no comments and proper variable names) and not efficiently implemented and minimal efforts shown	Program/code/simulation model/network model (i.e. with some comments but has improper variable names or vice versa) and is implemented by computationally complex routine	Program/code/simulation model/network model is well-documented (i.e. with comments and proper variable names) but not implemented efficiently	Program/code/simulation model/network model is well-documented (with comments and proper variable names) and task efficiently implemented in due time
LR3	Troubleshooting	Unable to identify the fault/minimal effort shown	Able to identify the fault but unable to remove it	Able to identify the fault but partially removes it	Able to identify the fault and able to make necessary steps and actions to correct it
LR4	Data Collection	Measurements are incomplete, inaccurate and imprecise. Observations are incomplete or not included. Symbols, units and significant figures are not included	Measurements are somewhat inaccurate and very imprecise. Observations are incomplete or recorded in a confusing way. Major errors using symbols, units and significant digits	Measurements are mostly accurate. Observations are generally complete. Minor errors using symbols, units and significant digits	Measurements are both accurate and precise. Observations are very thorough. Includes appropriate symbols, units and significant digits and task completed in due time
LR5	Results & Plots	Figures, graphs, tables contain errors or are poorly constructed, have missing titles, captions, units missing or incorrect, etc.	Most figures, graphs, tables OK, some still missing some important or required features	All figures, graphs, tables are correctly drawn, but some have minor problems or could still be improved	All figures, graphs, tables are correctly drawn and contain titles/captions
LR6	Clean-up	All equipment/PC is not powered off. All items left at station and not cleaned	Many equipment/PC is not powered off. Many items left at station	Some equipment/PC is not powered off. Some items left at station	All equipment/PC is powered off. Station left neat and clean
LR7	Viva	Response shows a complete lack of understanding of subjected task	Response shows some understanding of subjected task	Response shows substantial understanding of subjected task	Response shows complete understanding of subjected task
LR8	Contribution	No participation towards the group tasks	Slight participation towards the group tasks	Substantial participation towards the group tasks	Outstanding participation towards the group tasks
LR9	Report	No summary provided. The number/amount of tasks completed below the level of satisfaction and/or submitted late	Couldn't provide good summary of in-lab tasks. Some major tasks were completed but not explained well. Submission on time. Some major plots and figures provided	Good summary of In-lab tasks. All major tasks completed except few minor ones. The work is supported by some decent explanations, Submission on time, All necessary plots, and figures provided	Outstanding Summary of In-Lab tasks. All task completed and explained well, submitted on time, good presentation of plots and figure with proper label, titles and captions