MODULE 1

Overview of Digital Design with Verilog HDL: Evolution of CAD, emergence of HDLs, typical HDL-flow, why Verilog HDL?, trends in HDLs. Hierarchical Modeling Concepts: Top-down and bottom-up design methodology, differences between modules and module instances, parts of a simulation, design block, stimulus block.

OVERVIEW OF DIGITAL DESIGN WITH VERILOG HDL

1.1 Evolution of Computer-Aided Digital Design

In early days digital circuits were designed with vacuum tubes and transistor. Then integrated circuits chips were invented which consists of logic gates embed on them. As technology advances from SSI (Small Scale Integration), MSI (Medium Scale Integration), LSI (Large Scale Integration), designers could implement thousands of gates on a single chip. So the testing of circuits and designing became complicated hence Electronic Design Automation (EDA) techniques to verify functionality of building blocks were one.

The advances in semiconductor technology continue to increase the power and complexity of digital systems with the invent of VLSI (very Large Scale Integration) with more than 10000 transistors. Because of the complexity of circuit, breadboard design became impossible and gave rise to computer aided techniques to design and verify VLSI digital circuits. These computer aided programs and tools allow us to design, do automatic placement and routing and Abe to develop hierarchical based development and hence prototype development by downloading of programmable chips (like - ASIC, FPGA, CPLD) before fabrication.

1.2 Emergence of HDLs

In the field of digital design, the complexity in designing a circuit gave birth to standard languages to describe digital circuits (ie. Hardware Description Languages - HDL). HDL is a Computer Aided design (CAD) tool for the modern design and synthesis of digital systems. HDLs were been used to model hardware elements very concurrently. Verilog HDL and VHDL are most popular HDLs.

In initial days of HDL, designing and verification were done using tool but synthesis (i.e. translation of RTL to schematic circuit) used to be done manually which become tediously as technology advances. Later, tool is automated to generate the schematic of RTL developed.

Digital circuits are described at Registers Transfer Level (RTL) by using HDL. Then logic synthesis tool will generate details of gates and interconnection to implement circuits. This synthesized result can be used for fabrication by having placement and routing details. Verify functionality using simulation. HDLs are used for system-level design - simulation of system boards, interconnect buses, FPGAs and PALs. Verilog HDL is IEEE standard - IEEE 1364-2001.

1.3 Typical Design Flow

A typical design flow (HDL flow) for designing VLSI IC circuits is as shown in figure 1.1

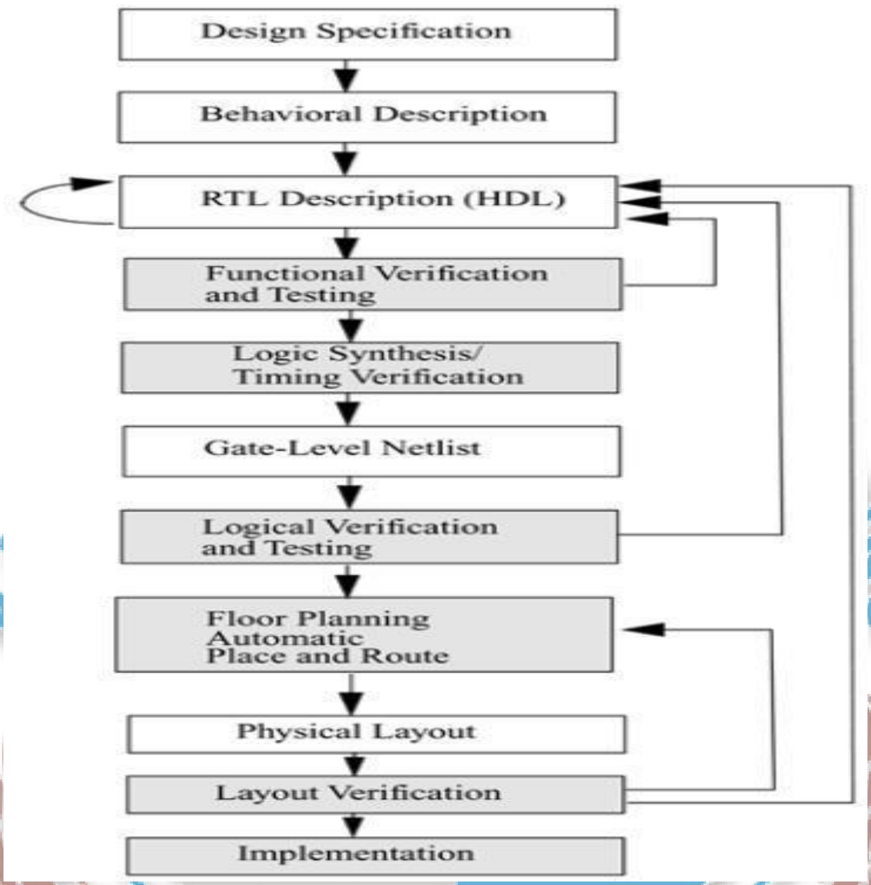
Design Specification
Behavioral Description
RTL Description (HDL)
Functional Verification and Testing
Logic Synthesis/ Timing Verification
Gate-Level Netlist
Logical Verification and Testing
Floor Planning Automatic Place and Route
Physical Layout
Layout Verification
Implementation

Figure: 1.1: Typical design flow

The design flow in any design, specifications are written first. Specifications describe abstractly the functionality, interface, and overall architecture of the digital circuit to be designed. At this point, the architects do not need to think about how they will implement this circuit. A behavioral description is then created to analyze the design in terms of functionality, performance, and compliance to standards, and other high-level issues. Behavioral descriptions are often written with HDLs.

New EDA tools have emerged to simulate behavioral descriptions of circuits. These tools combine the powerful concepts from HDLs and object oriented languages such as C++. These tools can be used instead of writing behavioral descriptions in Verilog HDL. The behavioral description is manually converted to an RTL description in an HDL. The designer has to describe the data flow that will implement the desired digital circuit. From this point onward, the design process is done with the assistance of EDA tools.

Logic synthesis tools convert the RTL description to a gate-level net list. Logic synthesis tools ensure that the gate-level net list meets timing, area, and power specifications. A gate-level net list is a description of the circuit in terms of gates and connections between them.

The gate-level net list is input to an Automatic Place and Route tool, which creates a layout. The layout is verified and then fabricated on a chip.

Thus, most digital design activity is concentrated on manually optimizing the RTL description of the circuit. After the RTL description is frozen, EDA tools are available to assist the designer in further processes. Designing at the RTL level has shrunk the design cycle times from years to a few months. It is also possible to do many design iterations in a short period of time.

Behavioral synthesis tools have begun to emerge recently. These tools can create RTL descriptions from a behavioral or algorithmic description of the circuit. As these tools mature, digital circuit design will become similar to high-level computer programming. Designers will simply implement the algorithm in an HDL at a very abstract level. EDA tools will help the designer convert the behavioral description to a final IC chip.

1.4 Importance of HDLs

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

- ❑ Designs can be described at a very 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 circuit. 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 early in the design cycle. Since designers work at the RTL level, 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 similar 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.

1.5 Popularity of Verilog HDL

Verilog HDL offers many useful features of hardware design.

- ❑ Verilog HDL is a general-purpose hardware description language that is easy to learn and easy to use. It is similar in syntax to the C programming language. Designers with C programming experience will find it easy to learn Verilog HDL.

- Verilog HDL allows different levels of abstraction to be mixed in the same model. Thus, a designer can define a hardware model in terms of switches, gates, RTL, or behavioral code. Also, a designer needs to learn only one language for stimulus and hierarchical design.
- Most popular logic synthesis tools support Verilog HDL. This makes it the language of choice for designers.
- All fabrication vendors provide Verilog HDL libraries for post-logic synthesis simulation. Thus, designing a chip in Verilog HDL allows the widest choice of vendors.
- The Programming Language Interface (PLI) is a powerful feature that allows the user to write custom C code to interact with the internal data structures of Verilog. Designers can customize a Verilog HDL simulator to their needs with the PLI.

1.6 Trends in HDLs

Increase in speed and complexity go digital circuits will complicate the designer job, but EDA tools make the job easy for designer. Designer has to do high level abstraction designing and need to take care of functionality of the design and EDA tools take care of implementation, and can achieve an almost optimum design.

Digital circuits are designed in HDL at an RTL level, so that logic synthesis tools can create gate level net lists. Behavioral synthesis allowed designers to directly design in terms of algorithms and the behavior of the circuit EDA tool is then used to translate and optimize at each phase of design. Verilog HDL is also used widely for verification.

Formal verification uses mathematical techniques to verify the correctness of Verilog HDL descriptions and to establish equivalency between RTL and gate level net lists. Assertion checking is done to check the transition and important parts of a design.

For very high-speed and timing critical circuits like microprocessors, designers mix gate-level description into the RTL description to achieve optimum results.

Another design technique is mixed bottom-up methodology where the designers use either existing Verilog HDL modules, basic building blocks, or vendor supplied core blocks to quickly bring up their system simulation, which helps to reduce development cost and compress design schedules.

1.7 Design Methodologies

There are two basic types of digital design methodologies: a top-down design methodology and a bottom-up design methodology.

1.7.1 Top-down design methodology: This designing approach allows early testing, easy change of different technologies, a well structures system design and offers many other advantages. In this method, top-level block is defined and sub-blocks necessary to build the top-level block are identified. We further subdivide, sub-blocks until cells cannot be further divided, we call these cells as leaf cells is as shown in figure 1.2.

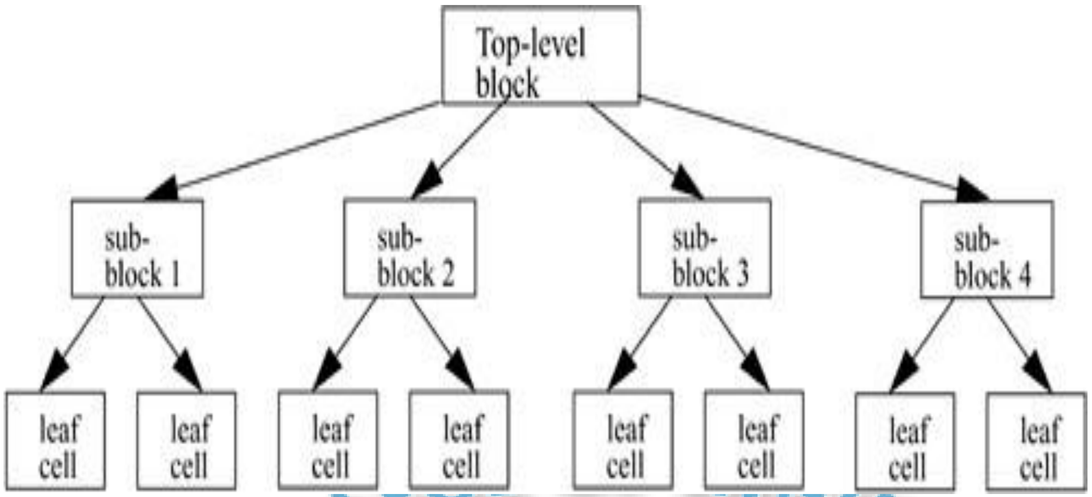

Figure: 1.2: Top-down Design Methodology

1.7.2 Bottom-up design methodology: We first identify the available building blocks and try to build bigger cells out of these, and continue process until we reach the top-level block of the design is as shown in figure 1.3

Most of the time, the combination of these two design methodologies are used to design. Logic designers decide the structure of design and break up the functionality into blocks and sub blocks. And designer will design a optimized circuit for leaf cell and using these will design top level design.

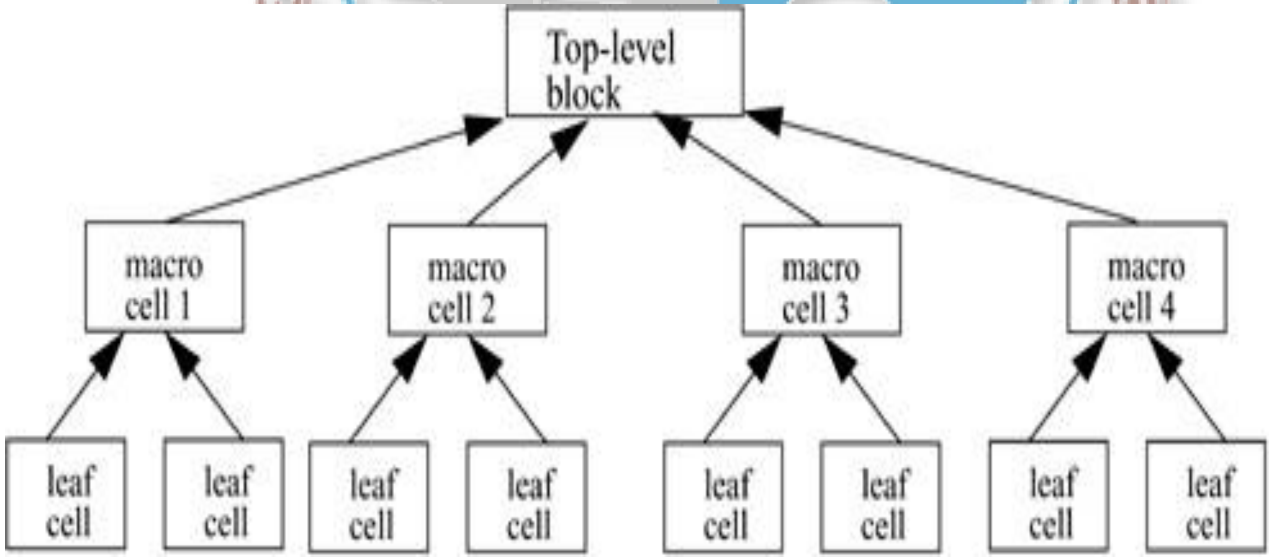

Figure 1.3: Bottom-up Design Methodology

A hierarchical modeling concept is illustrated with an example of 4-bit Ripple Carry Counter. The ripple carry counter shown in Figure 1.4 is made up of negative edge-triggered toggle flip-flops (T_FF). Each of the T_FFs can be made up from negative edge-triggered D-flip-flops (D_FF) and inverters (assuming q_bar output is not available on the D_FF), as shown in Figure 1.5.

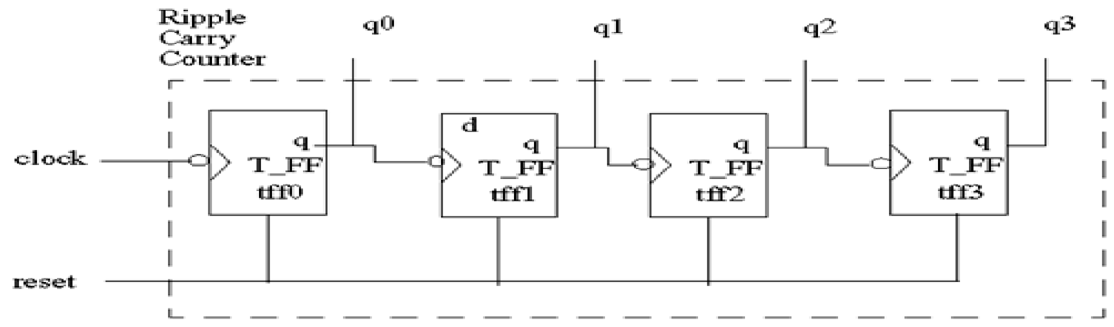

Figure 1.4: Ripple Carry Counter

reset	q_n	q_{n+1}
1	1	0
1	0	0
0	0	1
0	1	0
0	0	0

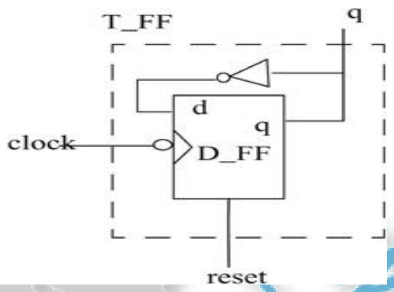

Figure 1.5: T-flip-flop

Thus, the ripple carry counter is built in a hierarchical fashion by using building blocks. The diagram for the design hierarchy is shown in Figure 1.6.

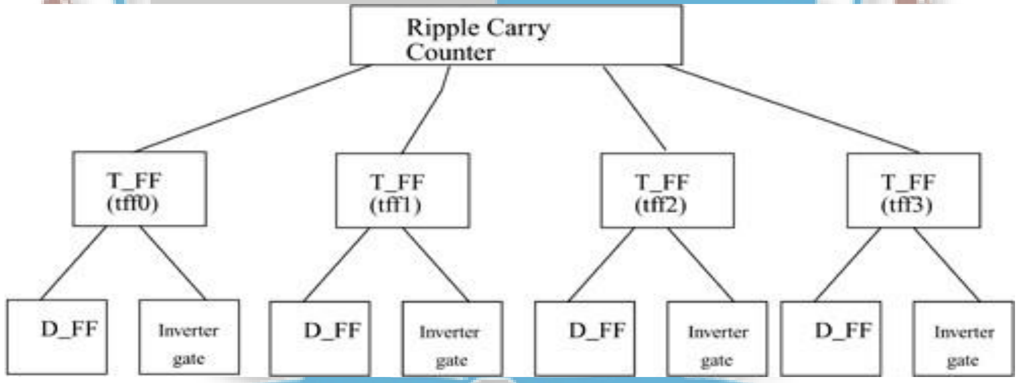

Figure 1.6: Design Hierarchy

In a top-down design methodology, we first have to specify the functionality of the ripple carry counter, which is the top-level block. Then, we implement the counter with T_FFs. We build the T_FFs from the D_FF and an additional inverter gate. Thus, we break bigger blocks into smaller building sub- blocks until we decide that we cannot break up the blocks any further.

A bottom-up methodology flows in the opposite direction. We combine small building blocks and build bigger blocks; e.g., we could build D_FF from and/ or gates, or we could build a custom D_FF from transistors. Thus, the bottom-up flow meets the top-down flow at the level of the D_FF.

1.8 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. Typically, elements are grouped into modules to provide common functionality that is used at many places in the design. A module provides the necessary functionality to the higher-level block 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.

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

Specifically, the T-flipflop could be defined as a module as follows:

```
module T_FF (q, clock, reset); .
.
<functionality of T-flipflop>
.
.
endmodule
```

Verilog is both a behavioral and a structural language. Internals of each module can be defined at four levels of abstraction, depending on the needs of the design. The levels are defined below.

- **Behavioral or algorithmic level:** This is the highest level of abstraction provided by Verilog HDL. A module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details. Designing at this level is very similar to C programming.
- **Dataflow level:** At this level, the module is designed by specifying the data flow. The designer is aware of how data flows between hardware registers and how the data is processed in the design.
- **Gate level:** The module is implemented in terms of logic gates and interconnections between these gates. Design at this level is similar to describing a design in terms of a gate-level logic diagram.

- **Switch level:** This is the lowest level of abstraction provided by Verilog. A module can be implemented in terms of switches, storage nodes, and the interconnections between them. Design at this level requires knowledge of switch-level implementation details.

Verilog allows the designer to mix and match all four levels of abstractions in a design.

1.9 Module 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 interface. The process of creating objects from a module template is called instantiation, and the objects are called instances.

In Example of 4 bit ripple carry counter, the top-level block creates four instances from the T-flipflop (T_FF) template. Each T_FF instantiates a D_FF and an inverter gate. Each instance must be given a unique name. Note that // is used to denote single-line comments.

Example of Module Instantiation

```
// Define the top-level module called ripple carry counter. It instantiates 4 T-flip-flops.
// Interconnections are shown in figure 1.4

module ripple_carry_counter(q, clk, reset);

output [3:0] q; //I/O signals and vector declarations

input clk, reset; //I/O signals

//Four instances of the module T_FF are created. Each has a unique name.

//Each instance is passed a set of signals. Each instance is a copy of the module T_FF.

T_FF tff0(q[0],clk, reset);

T_FF tff1(q[1],q[0], reset);

T_FF tff2(q[2],q[1], reset);

T_FF tff3(q[3],q[2], reset); endmodule

// Define the module T_FF. It instantiates a D-flipflop. Refer Figure 1-5 for interconnections.

module T_FF(q, clk, reset); output q;

input clk, reset;

wire d;

D_FF dff0(q, d, clk, reset); // Instantiate D_FF. Call it dff0.

not n1(d, q); // not gate is a Verilog primitive.

endmodule
```

In Verilog, it is illegal to nest modules. One module definition cannot contain another module definition within the module and endmodule statements.

Example below shows an illegal module nesting where the module T_FF is defined inside the module definition of the ripple carry counter.

Example for Illegal Module Nesting:

```
// Define the top-level module called ripple carry counter.

// It is illegal to define the module T_FF inside this module.

module ripple_carry_counter(q, clk, reset); output [3:0] q;

input clk, reset;

module T_FF(q, clock, reset);// ILLEGAL MODULE NESTING

…

<module T_FF internals>

…

endmodule // END OF ILLEGAL MODULE NESTING

endmodule
```

1.10 Components of a Simulation

Once a design block is completed, it must be tested. The functionality of the 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 blocks separate. The stimulus block can be written in Verilog. A separate language is not required to describe stimulus. The stimulus block is also commonly called a test bench.

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 signals clk and reset, and it checks and displays output signal q.

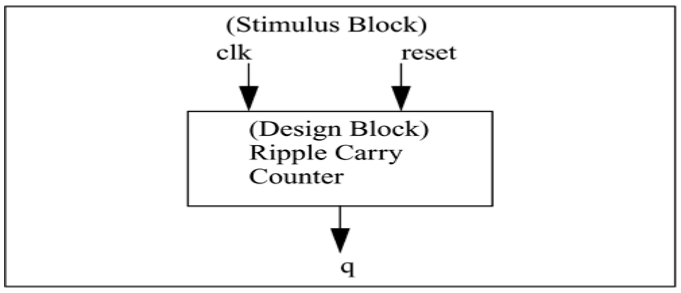

Figure 1.7: Stimulus Block Instantiates Design Block

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 signals d_clk and d_reset, which are connected to the signals clk and 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. Either stimulus style can be used effectively.

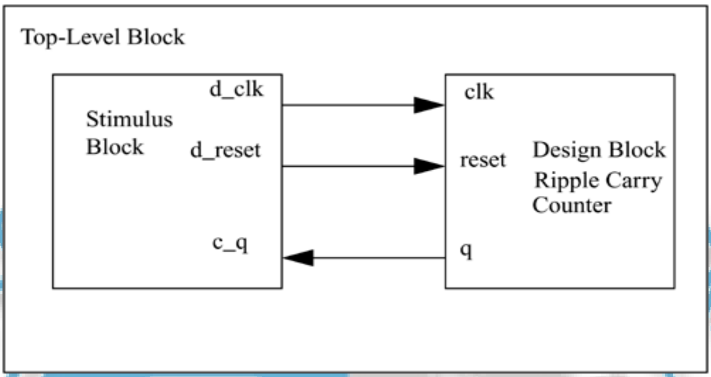

Figure 1.8: Stimulus Block and Design Block Instantiated in a dummy top level module

1.11 Example

Consider the example of simulation of a ripple carry counter. We will define the design block and the stimulus block. We will apply stimulus to the design block and monitor the outputs.

Design Block:

Consider a top-down design methodology. First, we write the Verilog description of the top-level design block which is the ripple carry counter.

Example of Ripple Carry Counter Top Block

```
module ripple_carry_counter(q, clk, reset);

output [3:0] q;

input clk, reset;

//4 instances of the module T_FF are created.

T_FF tff0(q[0],clk, reset); T_FF tff1(q[1],q[0], reset);

T_FF tff2(q[2],q[1], reset);

T_FF tff3(q[3],q[2], reset); endmodule
```

In the above module, four instances of the module T_FF (T-flipflop) are used. Therefore, we must now define the internals of the module T_FF.

Example for Flipflop T_FF

```
module T_FF(q, clk, reset);

output q;

input clk, reset; wire d;

D_FF dff0(q, d, clk, reset);

not n1(d, q); // not is a Verilog-provided primitive.

endmodule
```

Since T_FF instantiates D_FF, we must now define (Example 1-5) the internals of module D_FF. We assume asynchronous reset for the D_FFF.

Example for Flipflop D_F

```
// module D_FF with synchronous reset

module D_FF(q, d, clk, reset);

output q;

input d, clk, reset; reg q;

// Lots of new constructs. Ignore the functionality of the constructs.

// Concentrate on how the design block is built in a top-down fashion.

always@(posedge reset or negedge clk) if (reset)

q <= 1'b0;

else

q <= d;

endmodule
```

All modules have been defined down to the lowest-level leaf cells in the design methodology. The design block is now complete.

Stimulus Block:

We need to write the stimulus block to check if the ripple carry counter design 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 asynchronous reset mechanism are both tested. Consider the waveforms shown in Figure 1-9 to test the design. Waveforms for clk, reset, and 4-bit output q are shown. The cycle time for clk is 10 units; thereset signal stays up from time 0 to 15 and then goes up again from time 195 to 205. Output q counts from 0 to 15.

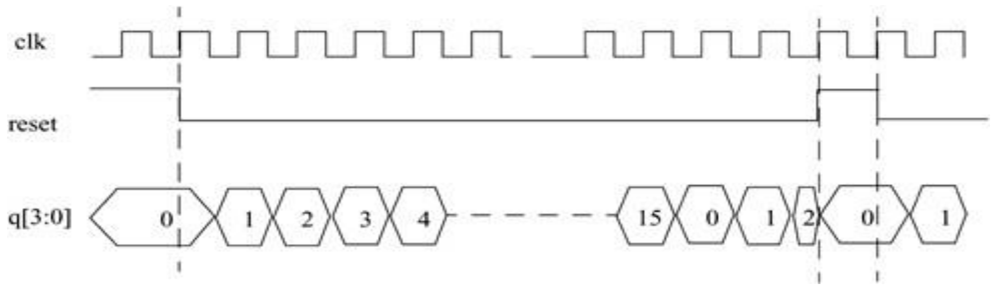

Figure 1.9: Stimulus and Output Waveforms

Example 1-6 Stimulus Block

module stimulus;

reg clk;

reg reset;

wire[3:0] q;

// instantiate the design block

ripple_carry_counter r1(q, clk, reset);

// Control the clk signal that drives the design block. Cycle time = 10

initial

clk = 1'b0; //set clk to 0 always

#5 clk = ~clk; //toggle clk every 5 time units

// Control the reset signal that drives the design block

// reset is asserted from 0 to 20 and from 200 to 220.

Initial

begin

reset = 1'b1;

#15 reset = 1'b0;

#180 reset = 1'b1;

#10 reset = 1'b0;

#20 $finish; *//terminate the simulation*

end

// Monitor the outputs

initial

```
$monitor($time, " Output q = %d", q);

endmodule
```

Once the stimulus block is completed, we are ready to run the simulation and verify the functional correctness of the design block.

Output of the Simulation:

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